First, We Will Complete Combinational Logic
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
Recall: Implementing a Full Adder Using a PLA

This input should not be connected to any outputs

We do not need this output

Truth table of a full adder

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>$c_{i+1}$</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>
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
  - E.g., Adder, Subtractor, Comparator, Shifter/Rotator, Multiplier, Divider
Comparator
Equality Checker (Compare if Equal)

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

**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>

Figure 5.14 ALU symbol
Example ALU (Arithmetic Logic Unit)

<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>

Table 5.1 ALU operations

![ALU Diagram]
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
  - E.g., Adder, Subtractor, Comparator, Shifter/Rotator, Multiplier, Divider
Tri-State Buffer
Tri-State Buffer

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

  ![Tri-state Buffer Diagram]

- Floating signal (Z): Signal that is not driven by any circuit
  - Open circuit, floating wire

Figure 2.40 Tristate buffer
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

```
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

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

**Figure 2.56** Multiplexer using tristate buffers
Recall: A 4-to-1 Multiplexer
Digging Deeper: Tri-State Buffer in CMOS

How do you implement Tri-State Buffers using transistors?
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>$c_i$</th>
<th>$c_{i+1}$</th>
<th>$s_i$</th>
<th>$c_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
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>

- If an input (B) can change without changing the output, that input value is not needed
  
  $F = A\overline{B} + AB = A(\overline{B} + B) = A(1) = A$

- $B$'s value changes within the rows where $F = 1$ (“ON set”)

- $A$'s value does NOT change within the ON-set rows

- $\rightarrow B$ 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>

- $B$'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 — \( F = A\overline{B} + AB \)

<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>

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

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!

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

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

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 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>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>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>0</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>0</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>0</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>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>1</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>1</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>1</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>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>1</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>1</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>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>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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<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>( X )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( X )</td>
<td>( X )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>( X )</td>
<td>( X )</td>
<td>( X )</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**...
  - We may cover them later if time permits

- 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/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=4570)
  - [https://youtu.be/ozs18ARNG6s?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=220](https://youtu.be/ozs18ARNG6s?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
Agenda for Today and Next Week

- **Today**
  - Start (and finish) Sequential Logic

- **Next week**
  - Hardware Description Languages and Verilog
    - Combinational Logic
    - Sequential Logic
  - Timing and Verification
Extra Credit Assignment 1: 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
Extra Credit Assignment 2: Moore’s Law

- **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 Credit Assignment 2: Moore’s Law

- 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
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
 Assignment: Readings

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

- **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

- **Timing and Verification**
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By the end of next week, make sure you are done with
  - P&P Chapters 1-3 + H&H Chapters 1-4
Readings (for Next Week)

- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full

- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By tomorrow, make sure you are done with
  - P&P Chapters 1-3 + H&H Chapters 1-4
Readings (for Next Next Week)

- **Von Neumann Model, LC-3, and MIPS**
  - P&P, Chapters 4, 5
  - H&H, Chapter 6
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)

- **Programming**
  - P&P, Chapter 6

- **Recommended:** Digital Building Blocks
  - H&H, Chapter 5
Sequential Logic Circuits and Design
What We Will Learn Today

- Circuits that can store information
  - Cross-coupled inverter
  - R-S Latch
  - Gated D Latch
  - D Flip-Flop
  - Register

- Finite State Machines (FSM)
  - State & Clock
  - Asynchronous vs. Synchronous
  - How to design FSMs
No Real Computer Can Work w/o Memory

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
A Large Fraction of Modern Systems is Memory

Apple M1 Ultra System (2022)

https://www.gsmarena.com/apple_announces_m1_ultra_with_20core_cpu_and_64core_gpu-news-53481.php
A Large Fraction of Modern Systems is Memory
A Large Fraction of Modern Systems is Memory

https://download.intel.com/newsroom/kits/40thanniversary/gallery/images/Pentium_4_6xx-die.jpg
A Large Fraction of Modern Systems is Memory

Core Count: 8 cores/16 threads
L1 Caches: 32 KB per core
L2 Caches: 512 KB per core
L3 Cache: 32 MB shared

AMD Ryzen 5000, 2020

Adding Even More Memory in 3D (2021)

AMD increases the L3 size of their 8-core Zen 3 processors from 32 MB to 96 MB

Additional 64 MB L3 cache die stacked on top of the processor die
- Connected using Through Silicon Vias (TSVs)
- Total of 96 MB L3 cache

https://youtu.be/gqAYMx34euU
https://www.tech-critter.com/amd-keynote-computex-2021/
A Large Fraction of Modern Systems is Memory

IBM POWER10, 2020

Cores:
15-16 cores, 8 threads/core

L2 Caches:
2 MB per core

L3 Cache:
120 MB shared

A Large Fraction of Modern Systems is Memory

Nvidia Ampere, 2020

Cores:
128 Streaming Multiprocessors

L1 Cache or Scratchpad:
192KB per SM
Can be used as L1 Cache and/or Scratchpad

L2 Cache:
40 MB shared
Cerebras’s Wafer Scale Engine-2 (2021)

- The largest ML accelerator chip
- 850,000 cores
- 40 GB of on-chip memory
- 20 PB/s memory bandwidth

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm²

https://cerebras.net/product/#overview

Largest GPU
54.2 Billion transistors
826 mm²
NVIDIA Ampere GA100
Circuits that Can Store Information
Introduction

- Combinational circuit output depends **only** on current input
- We want circuits that produce output depending on **current** and **past** input values – circuits with **memory**
- How can we design a circuit that **stores information**?
Capturing Data
Basic Element: Cross-Coupled Inverters

- Has two stable states: Q=1 or Q=0.
- Has a third possible “metastable” state with both outputs oscillating between 0 and 1 (we will see this later)
- Not useful without a control mechanism for setting Q

More Realistic Storage Elements

- **Have a control mechanism for setting Q**
  - We will see the R-S latch soon
  - Let’s look at an SRAM (static random access memory) cell first

![SRAM cell diagram]

- We will get back to SRAM (and DRAM) later
The Big Picture: Storage Elements

- **Latches and Flip-Flops**
  - Very fast, parallel access
  - Very expensive (one bit costs tens of transistors)

- **Static RAM (SRAM)**
  - Relatively fast
  - Expensive (one bit costs 6+ transistors)

- **Dynamic RAM (DRAM)**
  - Slower, reading destroys content (refresh), needs special process for manufacturing
  - Cheap (one bit costs only one transistor plus one capacitor)

- **Other storage technology (flash memory, hard disk, tape)**
  - Much slower, access takes a long time, non-volatile
  - Very cheap
Basic Storage Element:
The R-S Latch
The R-S (Reset-Set) Latch

- Cross-coupled **NAND gates**
  - Data is stored at Q (inverse at Q’)
  - S and R are control inputs
    - In *quiescent (idle) state*, both S and R are held at 1
    - **S (set):** drive S to 0 (keeping R at 1) to change Q to 1
    - **R (reset):** drive R to 0 (keeping S at 1) to change Q to 0
- S and R should never both be 0 at the same time

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Why not $R=S=0$?

1. If $R=S=0$, $Q$ and $Q'$ will both settle to 1, which breaks our invariant that $Q = \overline{Q'}$

2. If $S$ and $R$ transition back to 1 at the same time, $Q$ and $Q'$ begin to oscillate between 1 and 0 because their final values depend on each other (metastability)

- This eventually settles depending on variation in the circuits (more on this in the Timing Lecture)
The Gated D Latch
How do we **guarantee** correct operation of an R-S Latch?
The Gated D Latch

- How do we **guarantee** correct operation of an R-S Latch?
  - Add two more NAND gates!

- Q takes the value of D, when **write enable (WE)** is set to 1
- S and R can never be 0 at the same time!
The Gated D Latch

**Input** | **Output**
---|---
| |  
| WE | D | Q
| 0 | 0 | \( Q_{\text{prev}} \)
| 0 | 1 | \( Q_{\text{prev}} \)
| 1 | 0 | 0
| 1 | 1 | 1
The Register
How can we use D latches to store more data?
- Use more D latches!
- A single WE signal for all latches for simultaneous writes

Here we have a register, or a structure that stores more than one bit and can be read from and written to

This register holds 4 bits, and its data is referenced as Q[3:0]
The Register

How can we use D latches to store more data?
• Use more D latches!
• A single WE signal for all latches for simultaneous writes

Here we have a register, or a structure that stores more than one bit and can be read from and written to

This register holds 4 bits, and its data is referenced as Q[3:0]
Memory
Memory

- **Memory** is comprised of locations that can be written to or read from. An example memory array with 4 locations:

<table>
<thead>
<tr>
<th>Addr(00):</th>
<th>0100 1001</th>
<th>Addr(01):</th>
<th>0100 1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr(10):</td>
<td>0010 0010</td>
<td>Addr(11):</td>
<td>1100 1001</td>
</tr>
</tbody>
</table>

- Every unique location in memory is indexed with a unique **address**. 4 locations require 2 address bits ($\log[\#\text{locations}]$).

- **Addressability**: the number of **bits** of information stored in each location. This example: addressability is 8 bits.

- The entire set of **unique locations** in memory is referred to as the **address space**.

- Typical memory is **MUCH** larger (e.g., billions of locations)
Addressing Memory

Let’s implement a simple memory array with:
- 3-bit addressability & address space size of 2 (total of 6 bits)

1 Bit

6-Bit Memory Array

<table>
<thead>
<tr>
<th>Addr(0)</th>
<th>Bit₂</th>
<th>Bit₁</th>
<th>Bit₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr(1)</td>
<td>Bit₂</td>
<td>Bit₁</td>
<td>Bit₀</td>
</tr>
</tbody>
</table>
Reading from Memory

How can we select an address to read?
• Because there are 2 addresses, address size is $\log(2)=1$ bit
Reading from Memory

How can we select an address to read?
- Because there are 2 addresses, address size is $\log(2)=1$ bit

Addr[0]

Wordline

How can we select an address to read?

- Because there are 2 addresses, address size is $\log(2)=1$ bit
Reading from Memory

How can we select an address to read?
• Because there are 2 addresses, address size is $\log(2)=1$ bit
Recall: 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
How can we select an address and write to it?
How can we select an address and write to it?

- Input is indicated with $D_i$
Putting it all Together

Let’s enable reading from and writing to a memory array
A Bigger Memory Array (4 locations X 3 bits)
A Bigger Memory Array (4 locations X 3 bits)

Address Decoder

(toggether w/ decoder)
Figure 3.21  Reading location 3 in our $2^2$-by-3-bit memory.

Recall: 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)
Recall: A 4-to-1 Multiplexer
Aside: Implementing Logic Functions Using Memory
Recall: A Bigger Memory Array (4 locations X 3 bits)

Address Decoder

Multiplexer (together w/ decoder)
Memory-Based Lookup Table Example

- Memory arrays can also perform Boolean Logic functions
  - $2^N$-location $M$-bit memory can perform any $N$-input, $M$-output function
  - Lookup Table (LUT): Memory array used to perform logic functions
  - Each address: row in truth table; each data bit: corresponding output value

![Figure 5.52](Image)

4-word x 1-bit Array
## Lookup Tables (LUTs)

- LUTs are commonly used in FPGAs
  - To enable programmable/reconfigurable logic functions
  - To enable easy integration of combinational and sequential logic

### Table 1

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
<td>data 4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
<td>data 4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Figure 5.59** LE configuration for two functions of up to four inputs each

---

Read H&H Chapter 5.6.2
Recall: A Multiplexer-Based 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;
```

In Hardware (e.g., FPGA):

```
switch(input){
    case 0:
    case 1:
    case 2:
    case 4:
        return 0;
    default:
        return 1;
}
```
Sequential Logic Circuits
Sequential Logic Circuits

- We have examined designs of circuit elements that can *store information*
- Now, we will use these elements to build circuits that *remember* past inputs

**Combinational**
Only depends on current inputs

**Sequential**
Opens depending on past inputs

https://www.easykeys.com/228_ESP_Combination_Lock.aspx
https://www.fosmon.com/product/tsa-approved-lock-4-dial-combo
State

- In order for this lock to work, it has to keep track (remember) of the past events!
- If passcode is **R13-L22-R3**, sequence of **states** to unlock:
  A. The lock is not open (locked), and no relevant operations have been performed
  B. Locked but user has completed R13
  C. Locked but user has completed R13-L22
  D. Unlocked: user has completed R13-L22-R3

- The **state** of a system is a snapshot of all relevant elements of the system at the moment of the snapshot
  - To open the lock, **states A-D must be completed in order**
  - If anything else happens (e.g., L5), lock **returns** to state A
State Diagram of Our Sequential Lock

- Completely describes the operation of the sequential lock

We will understand “state diagrams” fully later today

Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an asynchronous “machine”
  - State transitions occur when they occur
  - There is nothing that synchronizes when each state transition must occur

- Most modern computers are synchronous “machines”
  - State transitions take place after fixed units of time
  - Controlled in part by a clock, as we will see soon

- These are two different design paradigms, with tradeoffs
Another Simple Example of State

- A standard Swiss traffic light has **4 states**
  - A. Green
  - B. Yellow
  - C. Red
  - D. Red and Yellow

- The sequence of these states are always as follows
  
  ![Sequence Diagram]

  A -> B -> C -> D
Changing State: The Notion of Clock (I)

- When should the light change from one state to another?
- We need a **clock** to dictate when to change state
  - Clock signal alternates between 0 & 1

![Diagram of states A, B, C, D connected with arrows]

CLK:

```
1
0
```

Figure 3.28  A clock signal.
When should the light change from one state to another?

We need a **clock** to dictate when to change state

- Clock signal alternates between 0 & 1

CLK:  

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Clock is a general mechanism that triggers transition from one state to another in a (synchronous) sequential circuit.

Clock synchronizes state changes across many sequential circuit elements.

Combinational logic evaluates for the length of the clock cycle.

Clock cycle should be chosen to accommodate maximum combinational circuit delay.

More on this later, when we discuss timing.
Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an asynchronous “machine”
  - State transitions occur when they occur
  - There is nothing that synchronizes when each state transition must occur

- Most modern computers are synchronous “machines”
  - State transitions take place after fixed units of time
  - Controlled in part by a clock, as we will see soon

- These are two different design paradigms, with tradeoffs
  - Synchronous control can be easier to get correct when the system consists of many components and many states
  - Asynchronous control can be more efficient (no clock overheads)

We will assume synchronous systems in most of this course
Finite State Machines
Finite State Machines

- **What is a Finite State Machine (FSM)?**
  - A discrete-time model of a stateful system
  - Each state represents a snapshot of the system at a given time

- An FSM pictorially shows
  1. the set of all possible **states** that a system can be in
  2. how the system transitions from one state to another

- An FSM can model
  - A traffic light, an elevator, fan speed, a microprocessor, etc.

- **An FSM enables us to pictorially think of a stateful system using simple diagrams**
Finite State Machines (FSMs) Consist of:

- **Five elements:**
  1. A **finite** number of **states**
     - **State:** snapshot of all relevant elements of the system at the time of the snapshot
  2. A **finite** number of external **inputs**
  3. A **finite** number of external **outputs**
  4. An explicit **specification of all state transitions**
     - How to get from one state to another
  5. An explicit **specification of what determines each external output value**
Each FSM consists of three separate parts:
- next state logic
- state register
- output logic

At the beginning of the clock cycle, next state is latched into the state register.
Finite State Machines (FSMs) Consist of:

- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge

- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be
  - Output logic
    - Generates the outputs
Finite State Machines (FSMs) Consist of:

- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Provide the next state at the clock edge

- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be
  - Output logic
    - Generates the outputs
State Register Implementation

- How can we implement a state register? Two properties:
  1. We need to store data at the beginning of every clock cycle
  2. The data must be available during the entire clock cycle

CLK: 0 1

Register Input: \[\text{Desired behavior}\]

Register Output:
The Problem with Latches

Currently, we cannot simply wire a clock to WE of a latch
- Whenever the clock is high, the latch propagates D to Q
- The latch is transparent

Recall the Gated D Latch

CLK: 1
0

Register Input:

Register Output:
Currently, we **cannot** simply wire a clock to WE of a latch

- **Whenever the clock is high**, the latch propagates $D$ to $Q$
- **The latch is transparent**

Recall the Gated D Latch

$$\begin{align*}
\text{CLK} &= \text{WE} \\
D &\rightarrow Q
\end{align*}$$

[Diagram of Gated D Latch]

**Undesirable!**
Currently, we cannot simply wire a clock to WE of a latch.

- When the clock is high, Q will not take on D's value AND
- When the clock is low, the latch will propagate D to Q.

The Problem with Latches

Recall the Gated D Latch

1) D (input) is observable at Q (output) only at the beginning of next clock cycle?

2) Q is available for the full clock cycle?
The Need for a New Storage Element

- To design viable FSMs

- We need storage elements that allow us to:
  - read the **current state** throughout the entire **current clock cycle**
  - not write the **next state** values into the storage elements until the beginning of the **next clock cycle**
The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

- When the clock is low, 1\textsuperscript{st} latch propagates \( D \) to the input of the 2\textsuperscript{nd} (Q unchanged)
- Only when the clock is high, 2\textsuperscript{nd} latch latches \( D \) (Q stores D)
  - At the rising edge of clock (clock going from 0->1), Q gets assigned D
The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

At the rising edge of clock (clock going from 0->1), $Q$ gets assigned $D$

At all other times, $Q$ is unchanged
The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

At the rising edge of clock (clock going from 0->1), \( Q \) gets assigned \( D \)

At all other times, \( Q \) is unchanged

We can use **D Flip-Flops** to implement the state register
Rising-Clock-Edge Triggered Flip-Flop

- **Two inputs:** CLK, D

- **Function**
  - The flip-flop "samples" D on the rising edge of CLK (*positive edge*)
  - When CLK rises from 0 to 1, D passes through to Q
  - Otherwise, Q holds its previous value
  - Q changes **only** on the rising edge of CLK

- A flip-flop is called an **edge-triggered state element** because it captures data on the clock edge
  - A latch is a **level-triggered** state element
We Covered
Until This Point
in Lecture
Slides for Future Lectures
D Flip-Flop Based Register

- Multiple parallel D flip-flops, each of which storing 1 bit

This register stores 4 bits

This line represents 4 wires

Condensed
A 4-Bit D-Flip-Flop-Based Register (Internally)

Finite State Machines (FSMs)

- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
  - **Moore FSM**: outputs depend only on the current state
Finite State Machines (FSMs)

- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
  - **Moore FSM**: outputs depend only on the current state
  - **Mealy FSM**: outputs depend on the current state and the inputs
Finite State Machine Example

- “Smart” traffic light controller
  - 2 inputs:
    - Traffic sensors: $T_A$, $T_B$ (TRUE when there’s traffic)
  - 2 outputs:
    - Lights: $L_A$, $L_B$ (Red, Yellow, Green)
- State can change every 5 seconds
  - Except if green and traffic, stay green

From H&H Section 3.4.1
Finite State Machine Black Box

- **Inputs**: CLK, Reset, $T_A$, $T_B$
- **Outputs**: $L_A$, $L_B$
**Finite State Machine Transition Diagram**

- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

- **S0**
  - \( L_A \): green
  - \( L_B \): red

- **Academic Ave.**
- **Bravado Blvd.**
- **Dining Hall**
- **Fields**
- **Labs**
- **Dorms**
- **Ave.**

- A Moore Finite State Machine (FSM) is characterized by outputs labeled in each state.
**Finite State Machine Transition Diagram**

- **Moore FSM**: outputs labeled in each state
  - **States**: Circles
  - **Transitions**: Arcs

Diagrams showing state transitions and outputs labeled in each state.
**Moore FSM:** outputs labeled in each state
- **States:** Circles
- **Transitions:** Arcs

Finite State Machine Transition Diagram

- **S0**
  - $L_A$: green
  - $L_B$: red
- **S1**
  - $L_A$: yellow
  - $L_B$: red
- **S2**
  - $L_A$: red
  - $L_B$: green

Reset

Academic Ave. ➔ Bravado Blvd. ➔ Dining Hall ➔ Labs ➔ Fields ➔ Dorms

T_A ➔ T_B ➔ T_A ➔ T_B ➔ T_B ➔ T_A ➔ Reset
Finite State Machine Transition Diagram

- **Moore FSM**: outputs labeled in each state
  - **States**: Circles
  - **Transitions**: Arcs

---

**S0**
- \( L_A \): green
- \( L_B \): red

**S1**
- \( L_A \): yellow
- \( L_B \): red

**S2**
- \( L_A \): red
- \( L_B \): green

**S3**
- \( L_A \): red
- \( L_B \): yellow

---

Reset

Academic Ave.

Bravado Blvd.

Dining Hall

Labs

Fields

Dorms

---
Finite State Machine Transition Diagram

- **Moore FSM**: outputs labeled in each state
  - **States**: Circles
  - **Transitions**: Arcs

```
S0
L_A: green
L_B: red

S1
L_A: yellow
L_B: red

S2
L_A: red
L_B: green

S3
L_A: red
L_B: yellow
```

- **Academic Ave.**
- **Bravado Blvd.**
- **Dining Hall**
- **Dorms**
- **Fields**
Finite State Machine: State Transition Table
FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

Legend:
- \( T_A \): green
- \( T_B \): red
- LA: light A
- LB: light B

Diagram:
- S0: \( L_A \): green, \( L_B \): red
- S1: \( L_A \): yellow, \( L_B \): red
- S2: \( L_A \): red, \( L_B \): green
- S3: \( L_A \): red, \( L_B \): yellow

Reset
FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td>S0</td>
<td>1</td>
<td>S0</td>
</tr>
<tr>
<td>S1</td>
<td>X</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>S3</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>S0</td>
</tr>
</tbody>
</table>
### FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$T_A$</td>
<td>$T_B$</td>
</tr>
<tr>
<td>S0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>S0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### State Encoding

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>S3</td>
<td>11</td>
</tr>
</tbody>
</table>
### FSM State Transition Table

**Current State** | **Inputs** | **Next State**
--- | --- | ---
\( S_1 \) | \( S_0 \) | \( T_A \) | \( T_B \) | \( S'_1 \) | \( S'_0 \)
-- | -- | -- | -- | -- | --
0 | 0 | 0 | X | 0 | 1
0 | 0 | 1 | X | 0 | 0
0 | 1 | X | X | 1 | 0
1 | 0 | X | 0 | 1 | 1
1 | 0 | X | 1 | 1 | 0
1 | 1 | X | X | 0 | 0

**State** | **Encoding**
--- | ---
S0 | 00
S1 | 01
S2 | 10
S3 | 11
### FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
<td>$T_A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

### State Encoding

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>S3</td>
<td>11</td>
</tr>
</tbody>
</table>
FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S0</td>
<td>T_A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

\[
S'_1 = (\overline{S}_1 \cdot S_0) + (S_1 \cdot \overline{S}_0 \cdot \overline{T}_B) + (S_1 \cdot \overline{S}_0 \cdot T_B)
\]

State Encoding:

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>S3</td>
<td>11</td>
</tr>
</tbody>
</table>
FSM State Transition Table

- **Current State**
  - **S0**: $L_A$: green, $L_B$: red
  - **S1**: $L_A$: yellow, $L_B$: red
  - **S2**: $L_A$: red, $L_B$: green
  - **S3**: $L_A$: red, $L_B$: yellow

- **Inputs**
  - $T_A$
  - $T_B$

- **Next State**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
<td>$T_A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

- **State Encoding**
  - **S0**: 00
  - **S1**: 01
  - **S2**: 10
  - **S3**: 11

- **Equation**
  - $S'_1 = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$
  - $S'_0 = ?$
**FSM State Transition Table**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( S_0 )</td>
<td>( T_A )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( X )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( X )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( X )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( X )</td>
</tr>
</tbody>
</table>

\[
S'_1 = (\overline{S}_1 \cdot S_0) + (S_1 \cdot \overline{S}_0 \cdot \overline{T}_B) + (S_1 \cdot S_0 \cdot T_B)
\]

\[
S'_0 = (\overline{S}_1 \cdot \overline{S}_0 \cdot \overline{T}_A) + (S_1 \cdot \overline{S}_0 \cdot \overline{T}_B)
\]

**State** | **Encoding**
---|---
\( S_0 \) | \( 00 \)
\( S_1 \) | \( 01 \)
\( S_2 \) | \( 10 \)
\( S_3 \) | \( 11 \)
### FSM State Transition Table

**State Encoding**

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>S3</td>
<td>11</td>
</tr>
</tbody>
</table>

**Current State**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S0</td>
<td>TA</td>
</tr>
<tr>
<td>S0</td>
<td>S1</td>
<td>TB</td>
</tr>
</tbody>
</table>

**Next State**

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>TA</th>
<th>TB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

**S' = S xor S0**  
(Simplified)

\[
S'_{0} = \overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A} + (S_1 \cdot S_0 \cdot \overline{T_B})
\]
Finite State Machine: Output Table
FSM Output Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
FSM Output Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>
**FSM Output Table**

**Current State**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$L_{A1}$</td>
<td>$L_{A0}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Output Encoding**

<table>
<thead>
<tr>
<th>Output</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>

$\text{Reset}$

$\text{L}_A = S_1$
FSM Output Table

Current State | Outputs
---|---
| $S_1$ | $S_0$ | $L_{A1}$ | $L_{A0}$ | $L_{B1}$ | $L_{B0}$
---|---|---|---|---|---
0 | 0 | 0 | 0 | 1 | 0
0 | 1 | 0 | 1 | 1 | 0
1 | 0 | 1 | 0 | 0 | 0
1 | 1 | 1 | 0 | 0 | 1

$L_{A1} = S_1$
$L_{A0} = \overline{S_1} \cdot S_0$

Output | Encoding
---|---
green | 00
yellow | 01
red | 10
# FSM Output Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
L_{A1} = S_1 \\
L_{A0} = S_1 \cdot S_0 \\
L_{B1} = S_1 \\
\]

<table>
<thead>
<tr>
<th>Output</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>
FSM Output Table

Current State | Outputs
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Output | Encoding
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>
Finite State Machine: Schematic
FSM Schematic: State Register
FSM Schematic: State Register

CLK

S'₁ S₁

S'₀ S₀

r

Reset

state register
FSM Schematic: Next State Logic

\[
\begin{align*}
S'_1 &= S_1 \text{ xor } S_0 \\
S'_0 &= (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})
\end{align*}
\]
FSM Schematic: Output Logic

\[ L_{A1} = S_1 \]
\[ L_{A0} = S_1 \cdot S_0 \]
\[ L_{B1} = S_1 \]
\[ L_{B0} = S_1 \cdot S_0 \]
FSM Timing Diagram

CLK
Reset
T_A
T_B
S'_1:0
S_1:0
L_A1:0
L_B1:0

S0
L_A: yellow
L_B: red

S1
L_A: yellow
L_B: red

S2
L_A: red
L_B: yellow

S3
L_A: red
L_B: green
FSM Timing Diagram

Cycle 1

CLK
Reset
T_A
T_B
S'_1:0
S_1:0
L_A: yellow
L_B: red
S0

S1
L_A: yellow
L_B: red

S2
L_A: red
L_B: green

S3
L_A: red
L_B: yellow

T_A
Reset
T_B
FSM Timing Diagram
FSM Timing Diagram

CLK
Reset
T_A
T_B
S'_1:0
S_1:0
L_A1:0
L_B1:0

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

S0
L_A: yellow
L_B: red

S1
L_A: yellow
L_B: red

S2
L_A: red
L_B: green

S3
L_A: red
L_B: yellow

S0
(00) 
S1
(01) 
S2

Green (00)
Red (10)
FSM Timing Diagram

- **S0**: $L_A$: yellow, $L_B$: red
  - $T_B$ transition
  - $T_A$ transition
- **S1**: $L_A$: yellow, $L_B$: red
- **S2**: $L_A$: red, $L_B$: green
- **S3**: $L_A$: red, $L_B$: yellow

**Clock Cycle Diagram**

- **CLK**
- **Reset**
- **$T_A$**
- **$T_B$**

**State Transitions**

- $S'_{1:0}$: state changes
- $S_{1:0}$: state changes
- $L_{A1:0}$: yellow, red
- $L_{B1:0}$: red, yellow

**Time Stamps**

- Cycle 1: $S_0$ (00)
- Cycle 2: $S_0$ (00)
- Cycle 3: $S_1$ (01)
- Cycle 4: $S_1$ (01)
- Cycle 5: $S_2$ (11)

**Time Markers**

- 0, 5, 10, 15, 20, 25 seconds
FSM Timing Diagram

- **Clock (CLK)**
- **Reset**
- **TA**
- **TB**
- **S'1:0**
- **S1:0**
- **LA1:0**
- **LB1:0**

**State Transitions:**
- **S0** (00) → **S1** (01) → **S2** (10) → **S3** (11)
- **S0** (00) → **S1** (01) → **S2** (10) → **S3** (11)

**Light States:**
- **LA**: Yellow
- **LB**: Red
- **LA**: Yellow
- **LB**: Green
- **LA**: Red
- **LB**: Yellow
- **LA**: Red
- **LB**: Yellow

**Time Cycles:**
- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7

**Timeline:**
- 0:00
- 5:00
- 10:00
- 15:00
- 20:00
- 25:00
- 30:00
- 35:00
- 40:00
- 45:00
This is from H&H Section 3.4.1
See H&H Chapter 3.4
Finite State Machine: State Encoding
How do we encode the state bits?

- Three common state binary encodings with different tradeoffs
  1. Fully Encoded
  2. 1-Hot Encoded
  3. Output Encoded

Let’s see an example **Swiss** traffic light with 4 states

- Green, Yellow, Red, Yellow+Red
FSM State Encoding (II)

1. Binary Encoding (Full Encoding):
   - Use the minimum possible number of bits
     - Use $\log_2(num\_states)$ bits to represent the states
   - Example state encodings: 00, 01, 10, 11
   - Minimizes # flip-flops, but not necessarily output logic or next state logic

2. One-Hot Encoding:
   - Each bit encodes a different state
     - Uses $num\_states$ bits to represent the states
     - Exactly 1 bit is “hot” for a given state
   - Example state encodings: 0001, 0010, 0100, 1000
   - Simplest design process – very automatable
   - Maximizes # flip-flops, minimizes next state logic
3. **Output Encoding:**

- Outputs are **directly accessible** in the state encoding.

- For example, since we have **3 outputs** (light color), encode state with **3 bits**, where each bit represents a color.

- **Example states:** 001, 010, 100, 110
  - Bit$_0$ encodes **green** light output,
  - Bit$_1$ encodes **yellow** light output
  - Bit$_2$ encodes **red** light output

- **Minimizes** output logic

- Only works for Moore Machines (output function of state)
3. Output Encoding:
   - Outputs are directly accessible in the state encoding
   
   The designer must carefully choose an encoding scheme to optimize the design under given constraints

   - Minimizes output logic
   - Only works for Moore Machines (output function of state)
Moore vs. Mealy Machines
Recall: Moore vs. Mealy FSMs

- Next state is determined by the current state and the inputs
- Two types of FSMs differ in the output logic:
  - **Moore FSM**: outputs depend only on the current state
  - **Mealy FSM**: outputs depend on the current state and the inputs
Moore vs. Mealy FSM Examples

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1’s and 0’s on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail’s brain.
Moore vs. Mealy FSM Examples

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1’s and 0’s on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail’s brain.
What are the tradeoffs?
FSM Design Procedure

- **Determine** all possible states of your machine

- **Develop** a *state transition diagram*
  - Generally this is done from a textual description
  - You need to 1) determine the **inputs** and **outputs** for each **state** and 2) figure out how to get from one state to another

- **Approach**
  - Start by defining the **reset state** and what happens from it – this is typically an easy point to start from
  - Then continue to add **transitions** and **states**
  - Picking **good state names** is very important
  - Building an FSM is **like** programming (but it *is not* programming!)
    - An FSM has a sequential “control-flow” like a program with conditionals and goto’s
    - The if-then-else construct is controlled by one or more inputs
    - The outputs are controlled by the state or the inputs
  - In hardware, we typically have many concurrent FSMs
What is to Come: LC-3 Processor

Figure 4.3 The LC-3 as an example of the von Neumann model
What is to Come: LC-3 Datapath
Backup Slides:
Different Types of Flip Flops
Enabled Flip-Flops

- **Inputs:** CLK, D, EN
  - The enable input (EN) controls when new data (D) is stored

- **Function:**
  - **EN = 1:** D passes through to Q on the clock edge
  - **EN = 0:** the flip-flop retains its previous state
Resettable Flip-Flop

- **Inputs**: CLK, D, Reset
  - The Reset is used to set the output to 0.

- **Function**:
  - **Reset = 1**: Q is forced to 0
  - **Reset = 0**: the flip-flop behaves like an ordinary D flip-flop

Symbols

![Flip-flop symbol](image_url)
Resettable Flip-Flops

- Two types:
  - **Synchronous**: resets at the clock edge only
  - **Asynchronous**: resets immediately when Reset = 1

- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop (see Exercise 3.10)

- Synchronously resettable flip-flop?
Settable Flip-Flop

- **Inputs:** CLK, D, Set
- **Function:**
  - Set = 1: Q is set to 1
  - Set = 0: the flip-flop behaves like an ordinary D flip-flop

Symbols

```
D  Q
Set
```

```
S
```
Backup Slides on Karnaugh Maps (K-Maps)
Complex Cases

- One example

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

- 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 $\leftrightarrow$ Logical adjacency

2-variable K-map

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td></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>000</td>
<td>001</td>
<td>011</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>101</td>
<td>111</td>
<td>110</td>
<td></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>0000</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>0101</td>
<td>0111</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1100</td>
<td>1101</td>
<td>1111</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>1001</td>
<td>1011</td>
<td>1010</td>
<td></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
K-map Cover - 4 Input Variables

Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:

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

\[ F = A + B\overline{D} + B\overline{C}D \]
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></th>
<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>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>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>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>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</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>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>0</td>
<td>1</td>
<td>0</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>
<td>0</td>
</tr>
<tr>
<td>1</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>1</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>1</td>
<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**

\[
\begin{array}{cccc}
A&B&C&D&\text{F1}&\text{F2}&\text{F3}\\
0&0&0&0&1&0&0\\
0&0&0&1&0&1&0\\
0&0&1&0&0&1&0\\
0&0&1&1&0&1&0\\
0&1&0&0&0&0&1\\
0&1&0&1&1&0&0\\
0&1&1&0&0&1&0\\
0&1&1&1&0&1&0\\
1&0&0&0&0&0&1\\
1&0&0&1&0&0&1\\
1&0&1&0&1&0&0\\
1&0&1&1&0&1&0\\
1&1&0&0&0&0&1\\
1&1&0&1&0&0&1\\
1&1&1&0&0&0&1\\
1&1&1&1&1&0&0\\
\end{array}
\]

F1 = 

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

K-map for F2

<table>
<thead>
<tr>
<th>F1</th>
<th>F2</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<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>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>

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>X</td>
<td>X</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>X</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>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

\[ Z \text{ (without don’t cares)} = \]

\[ Z \text{ (with don’t cares)} = \]
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