Transistor: Building Block of Computers

Microprocessors contain millions of transistors
- IBM PowerPC 750FX (2002): 38 million

Logically, each transistor acts as a switch
Combined to implement logic functions
- AND, OR, NOT
Combined to build higher-level structures
- Adder, multiplexer, decoder, register, ...
Combined to build processor
- LC-3
Simple Switch Circuit

Switch open:
• No current through circuit
• Light is off
• $V_{out}$ is +2.9V

Switch closed:
• Short circuit across switch
• Current flows
• Light is on
• $V_{out}$ is 0V

Switch-based circuits can easily represent two states: on/off, open/closed, voltage/no voltage.

n-type MOS Transistor

MOS = Metal Oxide Semiconductor
• two types: n-type and p-type

n-type
• when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
• when Gate has zero voltage, open circuit between #1 and #2 (switch open)

Terminal #2 must be connected to GND (0V).
p-type MOS Transistor

p-type is complementary to n-type

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)

Terminal #1 must be connected to +2.9V.

Logic Gates

Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.

Digital symbols:
- recall that we assign a range of analog voltages to each digital (logic) symbol

<table>
<thead>
<tr>
<th>Digital Values</th>
<th>0</th>
<th>“0”</th>
<th>Illegal</th>
<th>“1”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Values</td>
<td>0</td>
<td>0.5</td>
<td>2.4</td>
<td>2.9 Volts</td>
</tr>
</tbody>
</table>

- assignment of voltage ranges depends on electrical properties of transistors being used
  - typical values for "1": +5V, +3.3V, +2.9V
  - from now on we’ll use +2.9V
**CMOS Circuit**

**Complementary MOS**

Uses both **n-type** and **p-type** MOS transistors

- **p-type**
  - Attached to + voltage
  - Pulls output voltage UP when input is zero

- **n-type**
  - Attached to GND
  - Pulls output voltage DOWN when input is one

For all inputs, make sure that output is either connected to GND or to +, but not both!

---

**Inverter (NOT Gate)**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 V</td>
<td>2.9 V</td>
</tr>
<tr>
<td>2.9 V</td>
<td>0 V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth table
**NOR Gate**

Note: Serial structure on top, parallel on bottom.

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

**OR Gate**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>

Add inverter to NOR.
NAND Gate (AND-NOT)

Note: Parallel structure on top, serial on bottom.

AND Gate

Add inverter to NAND.
Rules of Boolean Algebra

1. Commutative Law
   - $A \cdot B = B \cdot A$
   - $A + B = B + A$

2. Associate Law
   - $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
   - $(A + B) + C = A + (B + C)$

3. Distributive Law
   - $(A + B) \cdot C = (A \cdot C) + (B \cdot C)$
   - $(A \cdot B) + C = (A + C) \cdot (B + C)$

4. Identities
   - $A + 0 = A$
   - $A \cdot 1 = A$

5. Inverse
   - $A + 1 = 1$
   - $A \cdot 0 = 0$

6. $A + A = A$
   - $A \cdot A = A$

7. $A + (A') = 1$
   - $A \cdot (A') = 0$

8. $A' = A$

9. De Morgan’s Theorem
   - $(A + B)' = (A)' \cdot (B)’$
   - $(A \cdot B)' = (A)' + (B)'$

Usually ' is evaluated first, then *, then +, with this order being changed by using parentheses.
DeMorgan's Law

Converting AND to OR (with some help from NOT)

Consider the following gate:

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

To convert AND to OR (or vice versa), invert inputs and output.

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

\[
\begin{align*}
(a + b + c)' &= a' b' c' \\
(a b c)' &= a' + b' + c' \\
(a b c) &= (a' + b' + c')'
\end{align*}
\]

Functionally complete set of gates

Any boolean function can be represented by the gates in this set:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR

Example: Convert the following boolean expression to a form that uses only gates in one of the above sets?

Hint: use De Morgan’s law

\[
f = abc + a'b'c + abc'
\]
Convert everything to NAND gates
The NAND gate is the universal gate. All other logic gates can be built from a NAND.

**NOT gate using NAND**

\[ \overline{A} = (A \cdot A) \text{ or } (A \cdot 1) \]

2-input AND gate using NAND gates

**AND gate using NAND**

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

Two steps
- First, compute \((A \cdot B)\)
- Next, invert the result using NOT gate (Slide 16) to get \((A \cdot B)\)
2-input OR gate using NAND gates

\[ Q = A + B = (\overline{A} \cdot \overline{B}) \text{ using De Morgan’s law} \]

Compute \( \overline{A} \) and \( \overline{B} \) using NOT operation

More than 2 Inputs?

AND/OR can take any number of inputs.

- AND = 1 if all inputs are 1.
- OR = 1 if any input is 1.
- Similar for NAND/NOR.

Can implement with multiple two-input gates, or with single CMOS circuit.
Canonical Forms

Standard form for a Boolean expression - unique algebraic expression directly from a true table (TT) description.

Two Types:
- Sum of Products (SOP)
- Product of Sums (POS)

Sum of Products: Output is 1 if any one of the input combinations that produce 1 is true. (disjunctive normal form, minterm expansion).

Example:

<table>
<thead>
<tr>
<th>minterms</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'b'c'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a'b'c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'bc'</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a'bc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ab'c'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ab'c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>abc'</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>abc</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

One product (and) term for each 1 in f:

\[
f = a'b'c' + ab'c' + ab'c + abc' + abc
\]

Alternate representation: Output is 1 if any none of the input combinations that produce 0 is true.

Example:

<table>
<thead>
<tr>
<th>minterms</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'b'c'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a'b'c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'bc'</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a'bc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ab'c'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ab'c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>abc'</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>abc</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

None of the zero terms are true =>

\[
f = (a'b'c')'(a'b'c)'(a'bc')'
\]

Using De Morgan’s law:

\[
f = (a + b + c)(a + b + c')(a + b' + c)
\]
Canonical Forms

**Product of Sums:** (conjunctive normal form, maxterm expansion).

**Example:**

<table>
<thead>
<tr>
<th>maxterms</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a+b+c'</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a+b'+c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a+b'+c'</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a'+b+c</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'+b+c'</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a'+b'+c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'+b'+c'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

One sum (or) term for each 0 in f:
f = (a+b+c)(a+b+c')(a+b'+c)

Mapping from SOP to POS (or POS to SOP): Derive truth table then proceed.

---

Sum of Products (cont.)

Canonical Forms are usually not minimal:

**Our Example:**

\[ f = a'bc + ab'c' + ab'c + abc' + abc + abc' + abc' + abc \]

\[ (xy' + xy = x) \]

\[ = a'bc + ab' + ab \]

\[ = a'bc + a \]

\[ (x'y + x = y + x) \]

\[ = a + bc \]

Goal: Simplify the boolean expression to use minimum number of gates
Karnaugh Maps
K-Maps are a convenient way to simplify Boolean Expressions.
They can be used for up to 4 or 5 variables.
They are a visual representation of a truth table.
Expression are most commonly expressed in sum of products form.

Truth table to K-Map

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

The expression is:
\[ \overline{A}.B + \overline{A}.B + A.B \]
minterms are represented by a 1 in the corresponding location in the K map.
K-Maps
Adjacent 1’s can be “paired off”
Any variable which is both a 1 and a zero in this pairing can be eliminated
Pairs may be adjacent horizontally or vertically
B is eliminated, leaving $\overline{A}$ as the term
A is eliminated, leaving B as the term
The expression becomes $\overline{A} + B$

Three Variable K-Map
One square filled in for each minterm.
Notice the code sequence: 00 01 11 10 – a Gray code.

$A \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$
Grouping the Pairs

Our truth table simplifies to \( A \cdot \overline{C} + B \cdot \overline{C} \).

Here, we can “wrap around” and this pair equates to \( A \cdot \overline{C} \) as \( B \) is eliminated.

The solution is \( B \) because it is a 1 over the whole block (vertical pairs) = \( BC + \overline{B}C = B(C + \overline{C}) = B \).
Karnaugh Maps

Three 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>A.B.C</td>
<td>A.B.C</td>
<td>A.B.C</td>
<td>A.B.C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A.B.C</td>
<td>A.B.C</td>
<td>A.B.C</td>
<td>A.B.C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Extreme ends of same row considered adjacent

Karnaugh Maps

Three Variable K-Map example

\[ X = \overline{A}.B.C + A.B.\overline{C} + \overline{A}.B.\overline{C} + A.B.\overline{C} \]

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

\[ X = \]
The Block of 4, again

\[
\begin{array}{c|c|c|c}
A & B & C & D \\
--- & --- & --- & --- \\
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
\end{array}
\]

\[X = \overline{C}\]

Karnaugh Maps

Four Variable K-Map example

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

\[
\begin{array}{c|c|c|c}
A & B & C & D \\
--- & --- & --- & --- \\
00 & 0 & 1 & 1 \\
01 & & & \\
11 & & & \\
10 & & & \\
\end{array}
\]
Karnaugh Maps

Four Variable K-Map solution

\[ F = \overline{A}.B.C.D + \overline{A}.B.C.D + \overline{A}.B.C.D + \overline{A}.B.C.D + \overline{A}.B.C.D + \overline{A}.B.C.D + \overline{A}.B.C.D \]

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

\[ F = \overline{B}.D + \overline{A}.C \]

Summary

MOS transistors are used as switches to implement logic functions.
- n-type: connect to GND, turn on (with 1) to pull down to 0
- p-type: connect to +2.9V, turn on (with 0) to pull up to 1

Basic gates: NOT, NOR, NAND
- Logic functions are usually expressed with AND, OR, and NOT

DeMorgan’s Law
- Convert AND to OR (and vice versa) by inverting inputs/output.
- Use to convert from Sum of Product to Product of Sum form.

Universal NAND Gates
- All other gates can be represented using NAND gates

Boolean Expressions
- Simplify boolean expression using Karnaugh maps
Building Functions from Logic Gates

**Combinational Logic Circuit**
- output depends only on the current inputs
- stateless

**Sequential Logic Circuit**
- output depends on the sequence of inputs (past and present)
- stores information (state) from past inputs

We'll first look at some useful combinational circuits, then show how to use sequential circuits to store information.

---

**Decoder**

$n$ inputs, $2^n$ outputs
- exactly one output is 1 for each possible input pattern

[Diagram of a 2-bit decoder with logic gates and conditions for each output]
**Multiplexer (MUX)**

*n*-bit selector and $2^n$ inputs, one output

- output equals one of the inputs, depending on selector

```
    A  B  C  D
  +---+---+---+---+
    |   |   |   |   |
  +---+---+---+---+
  |   |   |   |   |
  |   |   |   |   |
  +---+---+---+---+
    S_1  S_0

A, if S=00
B, if S=01
C, if S=10
D, if S=11
```

4-to-1 MUX

**Full Adder**

Add two bits and carry-in, produce one-bit sum and carry-out.

```
|   |   |   |   |   | S | C_out |
|---------------|-----|-----|
| A  | B  | C_in|   |   |   |      |
| 0  | 0  | 0   |   |   |   | 0 0   |
| 0  | 0  | 1   |   |   |   | 1 0   |
| 0  | 1  | 0   |   |   |   | 0 1   |
| 0  | 1  | 1   |   |   |   | 1 0   |
| 1  | 0  | 0   |   |   |   | 0 1   |
| 1  | 0  | 1   |   |   |   | 1 1   |
| 1  | 1  | 0   |   |   |   | 0 1   |
| 1  | 1  | 1   |   |   |   | 1 1   |
```

3-39

3-40
**Four-bit Adder**

![Four-bit Adder Diagram](image)

**Logical Completeness**

Can implement **ANY** truth table with AND, OR, NOT.

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

1. AND combinations that yield a "1" in the truth table.
2. OR the results of the AND gates.
Combinational vs. Sequential

Combinational Circuit
- always gives the same output for a given set of inputs
  - ex: adder always generates sum and carry, regardless of previous inputs

Sequential Circuit
- stores information
- output depends on stored information (state) plus input
  - so a given input might produce different outputs, depending on the stored information
- example: ticket counter
  - advances when you push the button
  - output depends on previous state
- useful for building “memory” elements and “state machines”

R-S Latch: Simple Storage Element
R is used to “reset” or “clear” the element – set it to zero.
S is used to “set” the element – set it to one.

If both R and S are one, out could be either zero or one.
- “quiescent” state – holds its previous value
- note: if a is 1, b is 0, and vice versa
Clearing the R-S latch
Suppose we start with output = 1, then change R to zero.

Output changes to zero.

Then set R=1 to "store" value in quiescent state.

Setting the R-S Latch
Suppose we start with output = 0, then change S to zero.

Output changes to one.

Then set S=1 to "store" value in quiescent state.
**R-S Latch Summary**

R = S = 1
- hold current value in latch

S = 0, R = 1
- set value to 1

R = 0, S = 1
- set value to 0

R = S = 0
- both outputs equal one
- final state determined by electrical properties of gates
- *Don’t do it!*

---

**Gated D-Latch**

Two inputs: D (data) and WE (write enable)

- when WE = 1, latch is set to value of D
  - S = NOT(D), R = D
- when WE = 0, latch holds previous value
  - S = R = 1

![Gated D-Latch Diagram]
Register
A register stores a multi-bit value.
• We use a collection of D-latches, all controlled by a common WE.
• When WE=1, n-bit value D is written to register.

Representing Multi-bit Values
Number bits from right (0) to left (n-1)
• just a convention -- could be left to right, but must be consistent
Use brackets to denote range:
D[1:r] denotes bit 1 to bit r, from left to right

A = \begin{array}{c}
\underline{0101001101010101} \\
\end{array}
\begin{array}{c}
^{15}_{0}
\end{array}

A[14:9] = 101001
A[2:0] = 101

May also see A<14:9>,
especially in hardware block diagrams.
Memory
Now that we know how to store bits, we can build a memory – a logical $k \times m$ array of stored bits.

**Address Space:**
number of locations
(usually a power of 2)

**Addressability:**
number of bits per location
(e.g., byte-addressable)

$2^2 \times 3$ Memory

![Diagram of a 2x3 memory with inputs and outputs labeled](image)
More Memory Details

This is not the way actual memory is implemented.
- fewer transistors, much more dense, relies on electrical properties

But the logical structure is very similar.
- address decoder
- word select line
- word write enable

Two basic kinds of RAM (Random Access Memory)

Static RAM (SRAM)
- fast, maintains data as long as power applied

Dynamic RAM (DRAM)
- slower but denser, bit storage decays – must be periodically refreshed

Also, non-volatile memories: ROM, PROM, flash, ...

State Machine

Another type of sequential circuit
- Combines combinational logic with storage
- "Remembers" state, and changes output (and state) based on inputs and current state

State Machine

Combinational Logic Circuit

Storage Elements

Inputs

Outputs
Combinational vs. Sequential
Two types of “combination” locks

**Combinational**
Success depends only on the **values**, not the order in which they are set.

**Sequential**
Success depends on the **sequence** of values (e.g., R-13, L-22, R-3).

State
The **state** of a system is a **snapshot** of all the relevant elements of the system at the moment the snapshot is taken.

Examples:
- The state of a basketball game can be represented by the scoreboard.
  - Number of points, time remaining, possession, etc.
- The state of a tic-tac-toe game can be represented by the placement of X’s and O’s on the board.
State of Sequential Lock

Our lock example has four different states, labelled A-D:

A: The lock is not open, and no relevant operations have been performed.

B: The lock is not open, and the user has completed the R-13 operation.

C: The lock is not open, and the user has completed R-13, followed by L-22.

D: The lock is open.

State Diagram

Shows states and actions that cause a transition between states.
Finite State Machine
A description of a system with the following components:

1. A finite number of states
2. A finite number of external inputs
3. A finite number of external outputs
4. An explicit specification of all state transitions
5. An explicit specification of what determines each external output value

Often described by a state diagram.
- Inputs trigger state transitions.
- Outputs are associated with each state (or with each transition).

The Clock
Frequently, a clock circuit triggers transition from one state to the next.

At the beginning of each clock cycle, state machine makes a transition, based on the current state and the external inputs.

- Not always required. In lock example, the input itself triggers a transition.
Implementing a Finite State Machine

Combinational logic
- Determine outputs and next state.

Storage elements
- Maintain state representation.

Storage: Master-Slave Flipflop
A pair of gated D-latches, to isolate next state from current state.

During 1st phase (clock=1), previously-computed state becomes current state and is sent to the logic circuit.

During 2nd phase (clock=0), next state, computed by logic circuit, is stored in Latch A.
Storage
Each master-slave flipflop stores one state bit.

The number of storage elements (flipflops) needed is determined by the number of states (and the representation of each state).

Examples:
• Sequential lock
  ➢ Four states – two bits
• Basketball scoreboard
  ➢ 7 bits for each score, 5 bits for minutes, 6 bits for seconds, 1 bit for possession arrow, 1 bit for half, ...

Complete Example
A blinking traffic sign
• No lights on
• 1 & 2 on
• 1, 2, 3, & 4 on
• 1, 2, 3, 4, & 5 on
• (repeat as long as switch is turned on)
Traffic Sign State Diagram

Transition on each clock cycle.

Traffic Sign Truth Tables

Outputs (depend only on state: \(S_1\)\(S_0\))

<table>
<thead>
<tr>
<th>(S_1)</th>
<th>(S_0)</th>
<th>(Z)</th>
<th>(Y)</th>
<th>(X)</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>1</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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Next State: \(S_1'S_0'\) (depend on state and input)

<table>
<thead>
<tr>
<th>(\text{In})</th>
<th>(S_1)</th>
<th>(S_0)</th>
<th>(S_1')</th>
<th>(S_0')</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Whenever \(\text{In}=0\), next state is 00.
Traffic Sign Logic

From Logic to Data Path

The data path of a computer is all the logic used to process information.
- See the data path of the LC-3 on next slide.

Combinational Logic
- Decoders -- convert instructions into control signals
- Multiplexers -- select inputs and outputs
- ALU (Arithmetic and Logic Unit) -- operations on data

Sequential Logic
- State machine -- coordinate control signals and data movement
- Registers and latches -- storage elements
LC-3 Data Path

Combinational Logic

Storage

State Machine