Logic

Why Study Logic?
- A basis for computer hardware
- A basis for computer programming
- A basis for program optimization
- A basis for specification
- A basis for verification and testing

In a certain sense

Computing is Logic

Is all Logic Computing?
No, but a lot of it can be reduced to computing.

Flavors of Logic
- Proposition Logic
- Predicate Logic  
- Temporal Logic
- Modal Logics
- Programming Logics
- Fuzzy Logic

Proposition Logic
- Also known as Switching Logic
- Basic elements are
  - 0 (false)
  - 1 (true)
  - proposition variables (take values 0 or 1)
  - either
    - functions (functional view)
    - connectives (expression view)

Studied in CS60
Some exposure in CS80
Some exposure in CS152 (Neural Networks)
Mostly we use

- the function view
- and occasionally
  - the expression view

Proposition Logic Domain

- \{false, true\} (for purists)
  - or
  - \{0, 1\} (more readable)
  - or
  - \{⊥, T\} (more symmetric)

Proposition Logic Functions

- and
- or
- not
- implies
- iff (if, and only if)
- others

and

shorter rex rules (using sequential convention):
- \(and(1, 1) \Rightarrow 1;\)
- \(and(x, y) \Rightarrow 0;\)
**and**

- form 2 table:

<table>
<thead>
<tr>
<th>and(x, y)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

results

**or**

- rex "table"

  - or(0, 0) => 0;
  - or(0, 1) => 1;
  - or(1, 0) => 1;
  - or(1, 1) => 1;

- shorter rex rules:

  - or(0, 0) => 0;
  - or(x, y) => 1;

**form 1 table**:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>or(x, y)</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>

**not**

- rex rules:

  - not(0) => 1;
  - not(1) => 0;
### not

<table>
<thead>
<tr>
<th>x</th>
<th>not(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### implies

#### Form 1 Table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>implies(x, 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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Form 2 Table:

<table>
<thead>
<tr>
<th>implies(x, y)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Rex Rules:

- `implies(0, 0) => 1;`
- `implies(0, 1) => 1;`
- `implies(1, 0) => 0;`
- `implies(1, 1) => 1;`

#### Shorter Rex Rules (Sequential):

- `implies(1, 0) => 0;`
- `implies(x, y) => 1;`

### Concise Summary

(Sequential convention applies)

- `and(1, 1) => 1;`
- `and(x, y) => 0;`
- `or(0, 0) => 0;`
- `or(x, y) => 1;`
- `not(0) => 1;`
- `not(1) => 0;`
- `implies(1, 0) => 0;`
- `implies(x, y) => 1;`
Expression Forms

- Use for greater readability of certain equalities
- Similar to ordinary discourse

Expression Forms

- Function symbols:
  - and: \( \land \) &\& juxtaposition
  - Example: \( x \land y \) means \( x \& y \)
  - or: \( \lor \) + ||
  - not: \( \neg \) ' superscript \( \overline{\cdot} \) (overbar)
- Example: These mean the same thing:
  - \( (a \land b) \lor (c \land \neg d) \)
  - \( ab + cd' \)
- Binding order: not, and, or

Expression Forms

- Function symbols:
  - implies: \( \rightarrow \) \( \supset \)
  - iff: \( \equiv \) ==

What We’ll Use

- To start, we’ll use
  \( \land \lor \neg \rightarrow \equiv \)
- When we discuss circuits, we’ll use
  \( + \cdot \) =

Logical Equivalences
(These can be shown by substituting all combinations of 0, 1 for variables)

- \( a \land b = b \land a \)
- \( a \lor b = b \lor a \)
- \( (a \land b) \land c = a \land (b \land c) \)
- \( (a \lor b) \lor c = a \lor (b \lor c) \)
- \( (a \lor b) \land c = (a \land c) \lor (b \land c) \)
- \( (a \land b) \lor c = (a \lor c) \land (b \lor c) \)

More Logical Equivalences

- \( (a \land 0) = 0 \)
- \( (a \land 1) = a \)
- \( (a \lor 0) = a \)
- \( (a \lor 1) = 1 \)
More Logical Equivalences

- \( \neg(a \land b) = (\neg b \lor \neg a) \)
- \( \neg(a \lor b) = (\neg b \land \neg a) \)

Logical Equivalences for Implies

- \( (a \rightarrow b) = (\neg a \lor b) \)
- \( (a \rightarrow b) = (\neg a \land \neg b) \)

DeMorgan’s Laws

- \( (a \lor \neg b) = a \lor b \)
- \( (a \land (\neg a \lor b)) = a \land b \)

Logical Equivalences for Implies

- \( (a \rightarrow bc) = (a \rightarrow b) \land (a \rightarrow c) \)
- \( ((a \rightarrow b) \land (b \rightarrow c)) \rightarrow (a \rightarrow c) \)
- \( (a \rightarrow b) = (\neg b \rightarrow \neg a) \)

Example

- Verify \( (a \rightarrow b) = (\neg b \rightarrow \neg a) \)
- Choose \( a \) as the variable.
  - Substituting 0 for \( a \): \( (0 \rightarrow b) = (\neg b \rightarrow \neg 0) \)
  - which simplifies to:
    - \( 1 = (\neg b \rightarrow 1) \), a known equivalence
  - Substituting 1 for \( a \):
    - \( (1 \rightarrow b) = (\neg b \rightarrow \neg 1) \)
    - which simplifies to:
      - \( b = (\neg b \rightarrow 0) \)

Checking Relations using the Boole–Shannon Principle

- Relations hold iff they hold for any substitution of 0 and 1 for the variables (uniformly throughout the expression)
- Therefore, a relation holds if, choosing any variable \( V \), it holds for \( V = 0 \) and for \( V = 1 \).
- But substituting 0 or 1 for a variable often yields simplifications that make the relation obvious.

Boole and Shannon

- Boole
  - Invented “Boolean algebra” (switching theory)
  - (In modern mathematics, “Boolean algebra” is a more general, abstract, system)
- Shannon
  - Wrote thesis on switching theory
  - Invented “Information theory”
  - Maze-solving mouse
  - Wrote first chess-playing program
  - Wrote paper on the mathematics of juggling
Boole and Shannon

George Boole (1815-1864) Claude Shannon (1916-2001)

Tautologies

- An expression that always evaluates to 1 (true) regardless of what value each variable is assigned is called a tautology.
- The property of being a tautology can be checked using:
  - Truth-table construction
  - Boole-Shannon Principle, recursively
- Example of a tautology checker (applet):

Encodings

- In order to use logic to build computers and other devices, we need to represent or encode general finite domains into the logic domain.

- At a sufficiently low-level, most information in a digital system is encoded into strings (or tuples) of bits.
- This may be true for the universe as a whole (e.g. Fredkin’s theory).

Encodings

- Let \( (0, 1)^n \) mean the set of all \( n \)-tuples of 0's and 1's, e.g.
- \( (0, 1)^3 = \{000, 001, 010, 011, 100, 101, 110, 111\} \)
- In general, an encoding of a set \( S \) is a function from \( S \) into \( (0, 1)^n \) for some \( n \) (the "number of bits").

Encoding Examples

(note: \( \rightarrow \) is maps to, not implies)

- Encode the set \{red, green, blue, black\}
  - Encoding #1:
    - red \( \rightarrow 00\), green \( \rightarrow 01\), blue \( \rightarrow 10\), black \( \rightarrow 11\)
  - Encoding #2:
    - red \( \rightarrow 01\), green \( \rightarrow 10\), blue \( \rightarrow 11\), black \( \rightarrow 00\)
  - Encoding #3 (called "one-hot" encoding)
    - red \( \rightarrow 0000\), green \( \rightarrow 0010\), blue \( \rightarrow 0011\), black \( \rightarrow 0001\)
Convention

- When the ordering is implied in the set itself (such as in a set of numbers), we will often omit the red part and just list the target value.
- Example: (red, green, blue, black) (00, 01, 10, 11)

How many bits are enough?

- To encode a set of size N, \[ \lceil \log_2(N) \rceil \] bits are needed, at a minimum.
- \( \lceil K \rceil \) is the smallest integer \( \geq K \)
  (read the "ceiling" of K).

Encoding Examples

- Encode the set \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}
  - Encoding #1 (straight binary encoding):
    - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
  - Encoding #2 (Gray encoding):
    - 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

Gray Code

- Invented by Frank Gray.
- Many important applications.

Gray Code: Based on "Reflection"

- 0 \rightarrow 1
  - Flip 1st (and only) bit

Gray Code: Based on "Reflection"

- Two copies of previous sequence
- 0 \rightarrow 1
- 0 \rightarrow 1
Gray Code: Based on "Reflection"

Two copies of previous sequence (black)
Reverse the second copy.
Prepend 0 to first, 1 to second.
00 → 01
↓
10 ← 11
Giving 00 → 01 → 11 → 10

Gray Code: Based on "Reflection"

Two copies of previous sequence (black)
Reverse the second copy.
Prepend 0 to first, 1 to second.
000 → 001 → 011 → 010
↓
100 ← 101 ← 111 ← 110

Going from one codeword to next entails changing only 1 bit.
Contrast to straight binary, where all bits but one could change.

Bonus: The code is cyclic.

Gray code generator in rex

```
grey(0) => [ ];
grey(n+1) => append(map(X) => [0 | X], grey(n)), map(X) => [1 | X], reverse(grey(n))));
grey(3) ➔
[[0, 0, 0], [0, 0, 1], [0, 1, 1], [0, 1, 0],
  [1, 1, 0], [1, 1, 1], [1, 0, 1], [1, 0, 0]]
```

Application: Shaft-Position Encoder

1-bit change property
Cyclic encoding:
No "glitches"
Example of a Commercial "Shaft Encoder"

Position information is provided as parallel Gray Code or Natural Binary, serial RS422, or 0-10V and/or 4-20mA analog outputs.

Available Configurations
FD-850 Incremental Digital
FD-850A 8 to 12 bit Absolutes

More Encoding Examples

- Encode the set \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
- Encoding #1 (BCD: binary-coded decimal)
  - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001
- Encoding #2 (7-segment encoding)
  - 1110111, 0010010, 1011101, 1011011, 0111010, 1101011, 1101111, 1010010, 1111111, 1111010

Encoding Examples

- Encoding #2 (7-segment encoding)
  - 1110111, 0010010, 1011101, 1011011, 0111010, 1101011, 1101111, 1010010, 1111111, 1111010

Error-Correcting Codes

- Information (bits)
  - 00101101
  - 00111101
  - 00101101

Unreliable communication (bits get changed)

- Error Correction
  - 0011101

Original Information (bits)

Error-Correcting Codes

- Uses:
  - Communication lines
  - Computer memory

The trick:
- Transmit extra bits
- But how?
- What if **those** bits are corrupted?

Error-Correcting Codes

- The objective:
  - Tolerate bit flips due to errors in communication, memory loss, etc.

The approach:
- Add redundant bits
  - If one bit is flipped, the original can still be recovered based on the resulting configuration

The clever part is that the "redundant" bits can also be flipped. The solution is **symmetric** about any code bit.

Hamming Code: Used for Error Correction
Richard W. Hamming, 1915-1998
worked with Shannon at Bell Labs

“The purpose of computing is insight, not numbers.”

Hamming Code Example

- Encode set {0, 1}
- **Attempt 1**: Use 1 extra bit:
  - 0 → 00, 1 → 11
  - **Not good enough**: 01 could be from either 0 or 1.
- **Attempt 2**: Use 2 extra bits:
  - 0 → 000, 1 → 111
  - **OK**: 001, 010, 100 all identify with 0
    - 011, 101, 110 all identify with 1

Hamming Code Visualization:
Hypercube

Hamming Code Visualization:
Hypercube

Efficiency

- The Hamming code may look inefficient (3 bits to encode 1 bit).
- However, it gets better as n gets larger.
- Among n bits total, only about log₂(n) are required to do support the error-correcting part.
- See text for larger example and how to compute Hamming code.

Analysis

- Let b be the total number of bits in the code.
- Let d be the number of data bits (2^d data words).
- For each data word, we need 1-b combinations for error correction (any of b bits could be flipped).
- Therefore 2^d(1-b) combinations in all are required, so 2^b > 2^d(1-b).
- size(d) = find(1, d);  
  find(b, d) = pow(2, b) >= pow(2, d)*(1+b) ? b : find(b+1, d);
- map[size, range(1, 20)];
  [3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25]
- e.g. 1 million code words can be handled with 25 bits.
5-Dimension Hypercube
2 data bits = 4 code words

Each different color represents the set of words that decode to the same data.
The black nodes are the codewords.

codewords: 0000 0001 1101 1110

distance is 3 between any two, as required for error correction.

Partially-Completed 6-Dimensional Hypercube

Visualize the missing connections. Find places for 8 code words.

7-bit Hamming Code
4 bits of data, 3 bits redundancy

7 bits = 2^7 combinations total, 2^4 of which are representatives.

Clever checksum technique for identifying which, if any, bit is wrong:

[b7 ⊕ b6 ⊕ b5 ⊕ b4, b7 ⊕ b6 ⊕ b3 ⊕ b2, b7 ⊕ b5 ⊕ b3 ⊕ b1] as binary
indices having 1 in high-order bit
indices having 1 in low-order bit

Logic Synthesis

Synthesizing Switching Functions

A "logic circuit" is composed of switching functions

Ways to Specify Switching Functions

- Logic circuit
- Functional expression
  - SOP form
  - Minterm expansion
- Truth table
- Compressed truth table
- BDD (Binary Decision Diagram)
- Index number
- Plot on a hypercube
- Plot on a Karnaugh map
SOP and Minterm forms

- This is called a SOP (sum-of-products) form. In this case, a "product" means product of variables or complemented variables.
- It is also a minterm form. A minterm is a product that uses all of the variables.
- Not every SOP is a minterm form.

Note the Connection

Read off each primed variable as 0, each unprimed as 1.

The "1" rows of the truth table correspond exactly to the minterms.

Shorthands

Show only the "1" rows (be careful)

Represent whole table by a set of "minterm numbers":
\{2, 5, 8\}

These are Equal

- The number of switching functions of \( n \) variables.
- The number of ways to assign 0 or 1 to the \( 2^n \) rows of the truth table.
- The number of subsets of \( \{0, 1, 2, ..., 2^n -1\} \)

Truth Table

<table>
<thead>
<tr>
<th>( u )</th>
<th>( v )</th>
<th>( w )</th>
<th>( x )</th>
<th>( f )</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>0</td>
<td>1</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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<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>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>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<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>
</tr>
<tr>
<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>0</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Represent whole table by a single numeral: 0010010010000000 = 934410
These are Equal

- The number of switching functions of $n$ variables.
- The number of ways to assign 0 or 1 to the $2^n$ rows of the truth table.
- The number of subsets of \{0, 1, 2, ..., $2^n - 1$\}
- $2^{2^n}$

Number of Switching Functions

- $2^{2^n}$
- $n = 1$: $2^2 = 4$
- $n = 2$: $2^4 = 16$
- $n = 3$: $2^8 = 256$
- $n = 4$: $2^{16} = 65,536$
- $n = 5$: $2^{32} = 4,294,967,296$
- $n = 6$: $2^{64} = 18,446,744,073,709,551,616$

The 16 switching functions of 2 variables

<table>
<thead>
<tr>
<th>args</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<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>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<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>xor</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>not</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>imp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logic Synthesis: Abstraction to Implementation

- From: Verbal problem description
- To: Implementation as a network of basic switching functions

Logic Synthesis: Stages

1. **Provide** verbal problem description.
2. **Tabulate** description as function on finite sets.
3. **Encode** finite sets into bits.
4. **Transcribe** the encoded tables.
5. **Split** into individual switching functions.
6. **Realize** as a network of basic gates.

Example

- Provide verbal description: Implement a “mod 3 adder using logic gates”
- A definition of mod3 addition:
  
  \[ f(a, b) = (a+b) \% 3; \]
  
  where $a, b \in \{0, 1, 2\}$
Tabulate definition of function

<table>
<thead>
<tr>
<th>(x+y)%3</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

 Encode sets into bits

- Set to be encoded: {0, 1, 2}
- Chosen encoding (among many):
  - 0 → 00
  - 1 → 01
  - 2 → 10

Transcribe the Function to the Encoded Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x+y)%3</td>
<td>0 1 2</td>
</tr>
<tr>
<td>0</td>
<td>0 1 2</td>
</tr>
<tr>
<td>1</td>
<td>1 2 0</td>
</tr>
<tr>
<td>2</td>
<td>2 0 1</td>
</tr>
</tbody>
</table>

Here first argument becomes u, second becomes v, etc.

Split the Encoded Function into individual switching functions

<table>
<thead>
<tr>
<th>Encoded Function</th>
<th>u  v  w  x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0 1 1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1 1 0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0 1 0 0</td>
</tr>
</tbody>
</table>

As the unspecified values will never occur, we can give the function either value 0 or 1.

For the time being, let’s just make them all 0.

Now we can “read off” an expression for each function.

<table>
<thead>
<tr>
<th>u  v  w  x</th>
<th>f₁</th>
<th>f₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
</tbody>
</table>
Exercise: "read off" the expression for $f_2$.

$$f_2(u, v, w, x) = u' v' w' x' + u' v w' x' + u v' w' x'$$

Realize each function by gates

Check by "Reverse Engineering" (Try all combinations; one combination is shown)

Rex Program for Checking all Combinations

The Testing Code