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

  } Studied in CS60

- Temporal Logic
- Modal Logics
- Programming Logics

  } Some exposure in CS80

- Fuzzy Logic

  } Some exposure in CS152 (Neural Networks)

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)

Mostly we use

- the function view
- and occasionally
  - the expression view

Proposition Logic Domain

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

Proposition Logic Functions

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

and

- rex "table":
  - and(0, 0) => 0;
  - and(0, 1) => 0;
  - and(1, 0) => 0;
  - and(1, 1) => 1;

and

- shorter rex rules (using sequential convention):
  - and(1, 1) => 1;
  - and(x, y) => 0;

and

- form 1 table:

x	y	and(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

arguments results

and

- form 2 table:

and(x, y)	0	1
0	0	0
1	0	1

results

or

- rex "table"
  - or(0, 0) => 0;
  - or(0, 1) => 1;
  - or(1, 0) => 1;
  - or(1, 1) => 1;

or

- shorter rex rules:
  - or(0, 0) => 0;
  - or(x, y) => 1;

or

- form 1 table:

x	y	or(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

or

- form 2 table:

or(x, y)	0	1
0	0	1
1	1	1

not

- rex rules:
  - not(0) => 1;
  - not(1) => 0;

not

- form 1 table = form 2 table:

x	not(x)
0	1
1	0

implies

- form 1 table:

x	y	implies(x, y)
0	0	1
0	1	1
1	0	0
1	1	1

implies

- form 2 table:

implies(x, y)	0	1
0	1	1
1	0	1

implies

- rex rules:
  - implies(0, 0) => 1;
  - implies(0, 1) => 1;
  - implies(1, 0) => 0;
  - implies(1, 1) => 1;

implies

- 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: $\wedge$ · && *juxtaposition*
    Example: x y means x $\wedge$y
  - or: $\vee$ + ||
  - not: $\neg$! ' superscript ‾(overbar)
- Example: These mean the same thing:
  - $(a \wedge b) \vee (c \wedge \neg d)$
  - 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
  $\wedge$ $\vee$ $\neg$ $\rightarrow$ $\equiv$
- When we discuss circuits, we'll use
  · + ' =

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

- $a \wedge b = b \wedge a$
- $a \vee b = b \vee a$
- $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
- $(a \vee b) \vee c = a \vee (b \vee c)$
- $(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$
- $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$

More Logical Equivalences

- $(a \wedge 0) = 0$
- $(a \wedge 1) = a$
- $(a \vee 0) = a$
- $(a \vee 1) = 1$

More Logical Equivalences

- $\neg(a \wedge b) = (\neg b \vee \neg a)$
- $\neg(a \vee b) = (\neg b \wedge \neg a)$

DeMorgan's Laws

- $(a \vee \neg a\, b) = a \vee b$
- $(a \wedge (\neg a \vee b)) = a \wedge b$

Logical Equivalences for Implies

- $(a \rightarrow b) = (\neg a \vee b)$
- $(a \rightarrow b) = \neg(a \wedge \neg b)$

- $(0 \rightarrow b) = 1$
- $(1 \rightarrow b) = b$
- $(a \rightarrow 0) = \neg a$
- $(a \rightarrow 1) = 1$

More Logical Equivalences for Implies

- $(a \rightarrow bc) = (a \rightarrow b) \wedge (a \rightarrow c)$
- $((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$
- $(a \rightarrow b) = (\neg b \rightarrow \neg a)$

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.

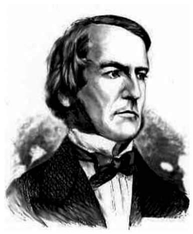
Example

- 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:
    - $b = (\neg b \rightarrow 0)$

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): http://www.cs.hmc.edu/~keller/javaExamples/taut/taut.html

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.

Encodings

- 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: → is *maps to*, not *implies*)

- Encode the set {red, green, blue, black}
  - Encoding #1:
    - red → 00, green →01, blue →10, black →11
  - Encoding #2:
    - red →01, green →10, blue →11, black →00
  - Encoding #3 (called **"one-hot" encoding**)
    - red →1000, green →0100, blue →0010 , black →0001

Convention

- When the ordering is implied in the set itself (such as in a set of numbers), we will often omit the

  red $\rightarrow$

  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.
- U.S. Patent 2 632 058, March 17, 1953.
- 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

Gray Code: Based on "Reflection"

0000 → 0001 → 0011 → 0010 → 0110 → 0111 → 0101 → 0100
↓
1000 ← 1001 ← 1011 ← 1010 ← 1110 ← 1111 ← 1101 ← 1100

Going from one codeword to next entails changing only 1 bit.

Contrast to straight binary, where all bits but one could change.

Gray Code: Based on "Reflection"

0000 → 0001 → 0011 → 0010 → 0110 → 0111 → 0101 → 0100
↑ ↓
1000 ← 1001 ← 1011 ← 1010 ← 1110 ← 1111 ← 1101 ← 1100

Bonus: The code is **cyclic**.

Gray code generator in rex

```
gray(0) => [[]];

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

gray(3) ➡

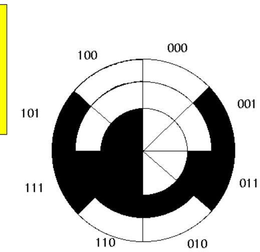
```
[[0, 0, 0], [0, 0, 1], [0, 1, 1], [0, 1, 0],
 [1, 1, 0], [1, 1, 1], [1, 0, 1], [1, 0, 0]]
```

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 ouputs.

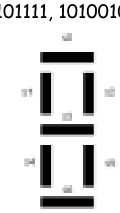
Available Configurations
FD-850 Incremental Digital
FD-850A 8 to 12 bit Absolutes

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

Unreliable communication
(bits get changed)

00111101

Error Correction

00101101

Original Information (bits)

Error-Correcting Codes

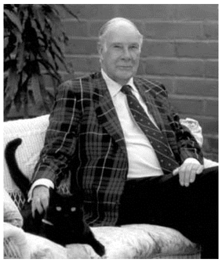
- Uses:
  - Communication lines
  - Computer memory
- The trick:
  - Transmit extra bits
  - But how?
  - What if *those* bits are corrupted?

Hamming Code: Used for Error Correction

- 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.

Richard W. Hamming, 1915-1998

worked with Shannon at Bell Labs

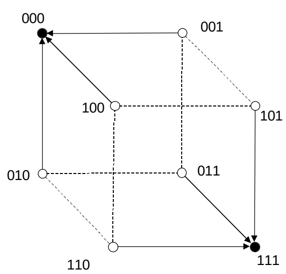


"The purpose of computing is insight, not numbers."

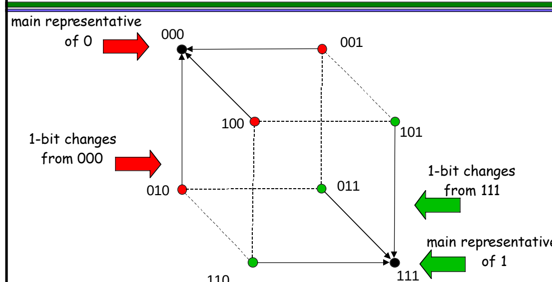
Hamming Code Example

- 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 $\log_2(n)$ 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 $2^b \geq 2^d(1+b)$.

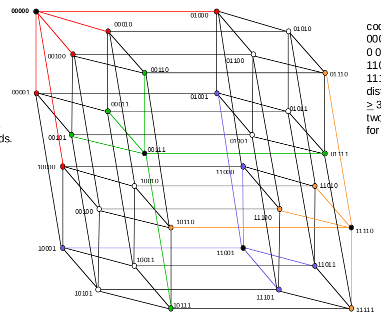
```
size(d) = find(1, d);
find(b, d) = pow(2, b) >= pow(2, d)*(1+b) ? b : find(b+1, d);
```

- 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

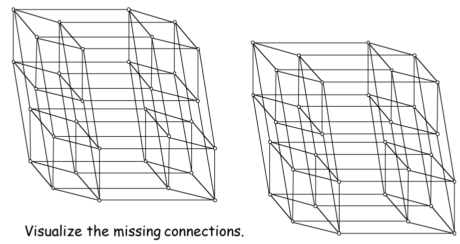
Each different color represents the set of words that decode to the same data.

The black nodes are the codewords.



codewords:
00000
00111
11001
11110
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.

		data	data	data	parity	data	parity	parity
decimal	**binary**	**b_7**	**b_6**	**b_5**	b_4	**b_3**	b_2	b_1
0	0000	**0**	**0**	**0**	0	**0**	0	0
1	0001	**0**	**0**	**0**	0	**1**	1	1
2	0010	**0**	**0**	**1**	1	**0**	0	1
3	0011	**0**	**0**	**1**	1	**1**	1	0
4	0100	**0**	**1**	**0**	1	**0**	1	0
5	0101	**0**	**1**	**0**	1	**1**	0	1
6	0110	**0**	**1**	**1**	0	**0**	1	1
7	0111	**0**	**1**	**1**	0	**1**	0	0
8	1000	**1**	**0**	**0**	1	**0**	1	1
9	1001	**1**	**0**	**0**	1	**1**	0	0
10	1010	**1**	**0**	**1**	0	**0**	1	0
11	1011	**1**	**0**	**1**	0	**1**	0	1
12	1100	**1**	**1**	**0**	0	**0**	0	1
13	1101	**1**	**1**	**0**	0	**1**	1	0
14	1110	**1**	**1**	**1**	1	**0**	0	0
15	1111	**1**	**1**	**1**	1	**1**	1	1

These are the representatives.

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

$[\underbrace{b_7 \oplus b_6 \oplus b_5 \oplus b_4}_{\text{indices having 1 in high-order bit}},\ \underbrace{b_7 \oplus b_6 \oplus b_3 \oplus b_2},\ \underbrace{b_7 \oplus b_5 \oplus b_3 \oplus b_1}_{\text{indices having 1 in low-order bit}}]$ as binary

$\oplus$ is xor.

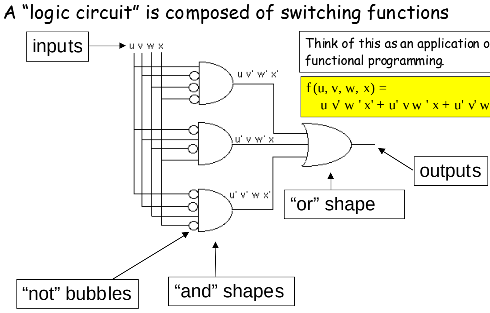
Logic Synthesis

Synthesizing Switching Functions

A "logic circuit" is composed of switching functions

Think of this as an application of functional programming.

f (u, v, w, x) =
u v' w ' x' + u' v w ' x + u' v' w x'



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

These are seen on the previous page.

SOP and Minterm forms

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

- 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.

Truth Table

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

u v w x	f
0 0 0 0	0
0 0 0 1	0
0 0 1 0	1
0 0 1 1	0
0 1 0 0	0
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	0
1 0 1 0	0
1 0 1 1	0
1 1 0 0	0
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Note the Connection

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

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

u v w x	f
0 0 0 0	0
0 0 0 1	0
0 0 1 0	1
0 0 1 1	0
0 1 0 0	0
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	0
1 0 1 0	0
1 0 1 1	0
1 1 0 0	0
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Note the Connection

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

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

u v w x	f
0 0 0 0	0
0 0 0 1	0
0 0 1 0	1
0 0 1 1	0
0 1 0 0	0
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	0
1 0 1 0	0
1 0 1 1	0
1 1 0 0	0
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Shorthands

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

Show only the "1" rows (be careful)

u v w x	f
0 0 1 0	1
0 1 0 1	1
1 0 0 0	1

Represent whole table by a set of "minterm numbers":

{2, 5, 8}

Represent whole table by a single numeral: $0010010010000000 = 9344_{10}$

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 $\{0, 1, 2, \ldots, 2^n - 1\}$

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
  $\{0, 1, 2, \ldots, 2^n - 1\}$
- 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: $2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$

Each level **squares** the previous, since

$$2^{2^{n+1}} = 2^{2 \cdot 2^n} = 2^{2^n + 2^n} = (2^{2^n})^2$$

The 16 switching functions of 2 variables

args		Function number															
b	**c**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	**0**	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	**1**	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	**0**	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	**1**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
			and					**xor**	**or**	**nor**	**iff**				**implies**	**nand**	

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

form 2 table:

(x+y)%3	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Encode sets into bits

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

Transcribe the Function to the Encoded Values

Function

(x+y)%3	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Encoding

$0 \rightarrow 00$
$1 \rightarrow 01$
$2 \rightarrow 10$

Here first argument becomes uv, second becomes xy.

Encoded Function

xy \ uv	00	01	10
00	00	01	10
01	01	10	00
10	10	00	01

Split the Encoded Function into individual **switching** functions

Encoded Function

xy \ uv	00	01	10
00	0 0	0 1	1 0
01	0 1	1 0	0 0
10	1 0	0 0	0 1

left bit

right bit

Call this f_1.

xy \ uv	00	01	10
00	0	0	1
01	0	1	0
10	1	0	0

Call this f_2.

xy \ uv	00	01	10
00	0	1	0
01	1	0	0
10	0	0	1

The resulting switching functions generally will only be ***partially*** specified; some combinations don't occur.

f_1

xy \ uv	00	01	10
00	0	0	1
01	0	1	0
10	1	0	0

f_2

xy \ uv	00	01	10
00	0	1	0
01	1	0	0
10	0	0	1

u	v	w	x	f_1	f_2
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?

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.

$f_1(u, v, w, x) =$
$u' v' w x'$
$+ u' v w' x$
$+ u v' w' x'$

u	v	w	x	f_1	f_2
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?

Exercise: "read off" the expression for f_2.

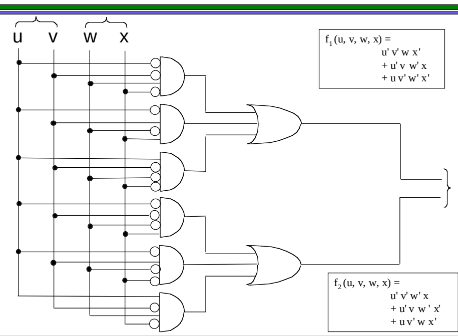
u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	0 0
1 0 0 1	0 0
1 0 1 0	1 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

Exercise: "read off" the expression for f_2.

u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	0 0
1 0 0 1	0 0
1 0 1 0	1 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

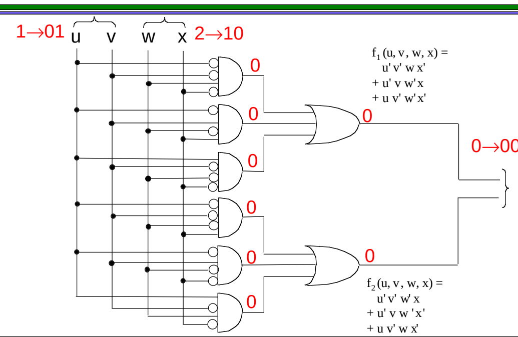
$f_2(u, v, w, x) =$
$u' v' w' x$
$+ u' v w' x'$
$+ u v' w x'$

Realize each function by gates



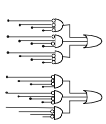
Check by "Reverse Engineering"

(Try all combinations; one combination is shown)



Rex Program for Checking all Combinations

```
encode(0) => [0, 0];
encode(1) => [0, 1];
encode(2) => [1, 0];

decode([0, 0]) => 0;
decode([0, 1]) => 1;
decode([1, 0]) => 2;
decode([x, y]) => "error, should not occur";

add3(i, j) = (i+j)%3;



addByCircuit(i, j) = decode(circuit(encode(i), encode(j)));

circuit([u, v], [w, x]) =>
    [   !u && !v &&  w && !x
     || !u &&  v && !w &&  x
     ||  u && !v && !w && !x,

        !u && !v && !w &&  x
     || !u &&  v && !w && !x
     ||  u && !v &&  w && !x
    ];
```



The Testing Code

```
test(addByCircuit(0, 0), add3(0, 0));
test(addByCircuit(0, 1), add3(0, 1));
test(addByCircuit(0, 2), add3(0, 2));

test(addByCircuit(1, 0), add3(1, 0));
test(addByCircuit(1, 1), add3(1, 1));
test(addByCircuit(1, 2), add3(1, 2));

test(addByCircuit(2, 0), add3(2, 0));
test(addByCircuit(2, 1), add3(2, 1));
test(addByCircuit(2, 2), add3(2, 2));
```