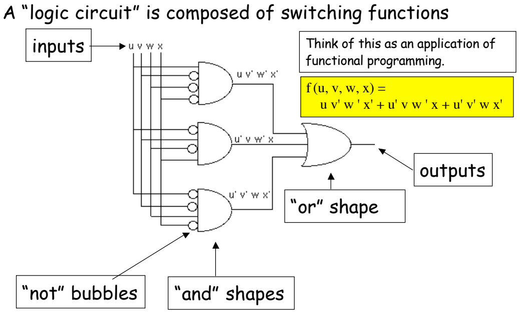
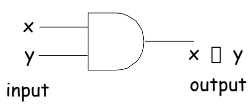
Logic Circuit Synthesis

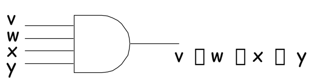
Synthesizing Switching Functions

A "logic circuit" is composed of switching functions

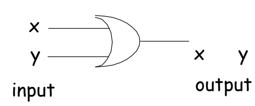


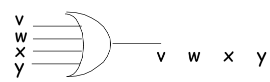
"and" gates



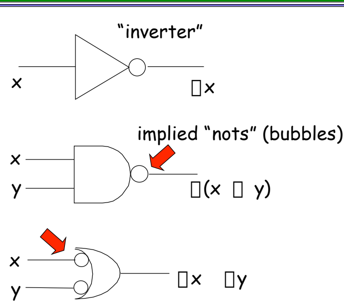


"or" gates





"not" Gates



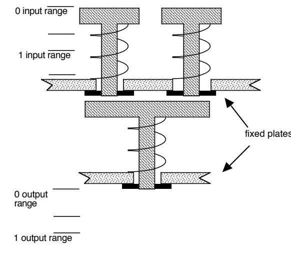
"Bill" Gates

Gates are Symbolic

- Gates are not just electronic; they can be
  - Mechanical
  - Hydraulic ("fluid logic")
  - Biological
  - Sub-atomic
  - Quantum-mechanical

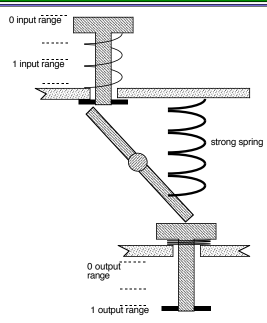
Mechanical Gates

"or" gate



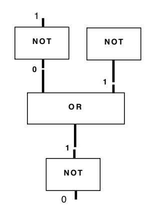
Mechanical Gates

"inverter"



Mechanical Gates

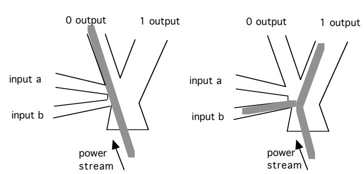
"and" gate



"Nanotechnology"

Fluid Gates

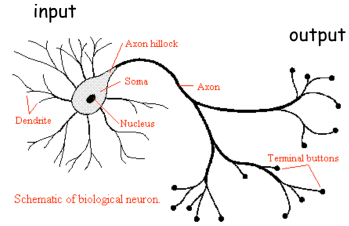
"or" gate



Biological Gates

input

output



DNA-based Gates

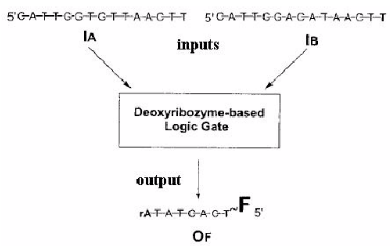
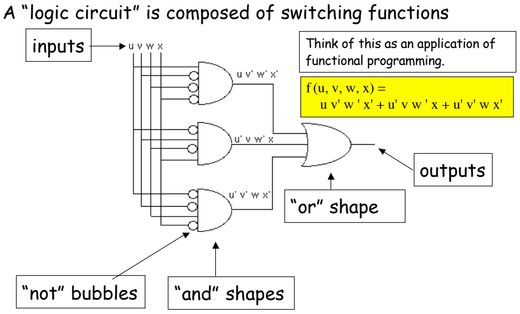


Figure 8: schematic of a logic deoxyribozyme-based logic gate with the input and output oligonucleotides.

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

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.

Truth table

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

Definition of "minterm"

- In the context of an n-variable switching function, a minterm is a function that is a conjunction ("and") of each variable or its complement (but not both).
- Minterm Examples (4 variables: u, v, w, x):
  uv'w'x
  u'vw'x
- Non-Minterm Examples (4 variables):
  uv
  x
  uvw
  u'uvw

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}

Number of Switching Functions

- How many switching functions of n variables are there?
- This will be on the final, so might as well memorize it now.

These are Equal

- The number of switching functions of **n variables**.
- The number of ways to assign 0 or 1 to the 2^n combinations of n variables.
- 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: $2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$

Each level **squares** the previous, since

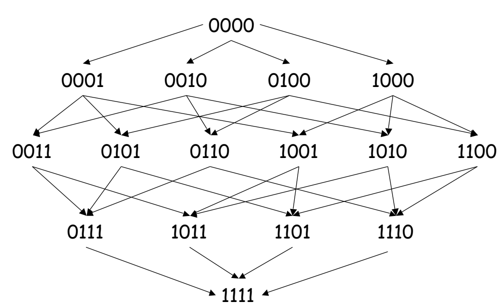
$$2^{2^{n+1}} = 2^{2\cdot 2^n} = 2^{2^n+2^n} = (2^{2^n})^2$$

- Also remember that 2^{10} = 1024 approx. 1000
  2^{20} = approx. 1,000,000 etc.

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
		constant 0	and	□implies	projection	□implies reversed	projection	xor	or	nor	iff	□projection	implies reversed	□projection	implies	nand	constant 1

Implication Lattice of 2-variable functions



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 ∈ {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 → 00
  - 1 → 01
  - 2 → 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 → 00
1 → 01
2 → 10

Here first argument becomes uv, second becomes xy.

Encoded Function

		uv		
		00	01	10
xy	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.

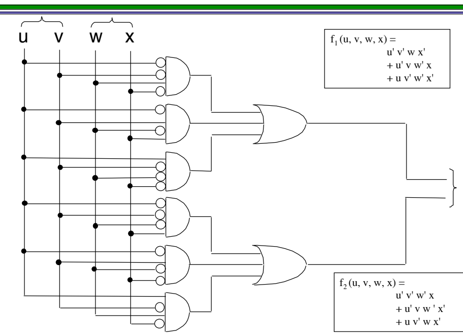
u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	0 0
1 0 0 1	0 0
1 0 1 0	1 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

Exercise: "read off" the expression for f_2.

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

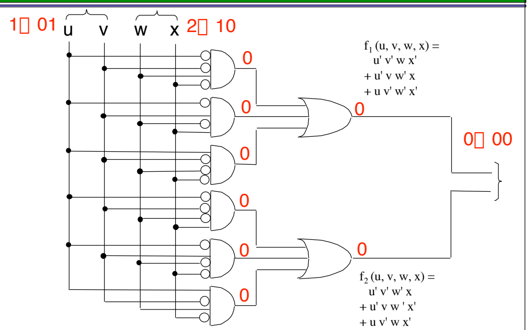
u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	0 0
1 0 0 1	0 0
1 0 1 0	1 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

Realize each function by gates



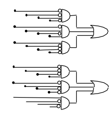
Check by "Reverse Engineering"

(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));
```