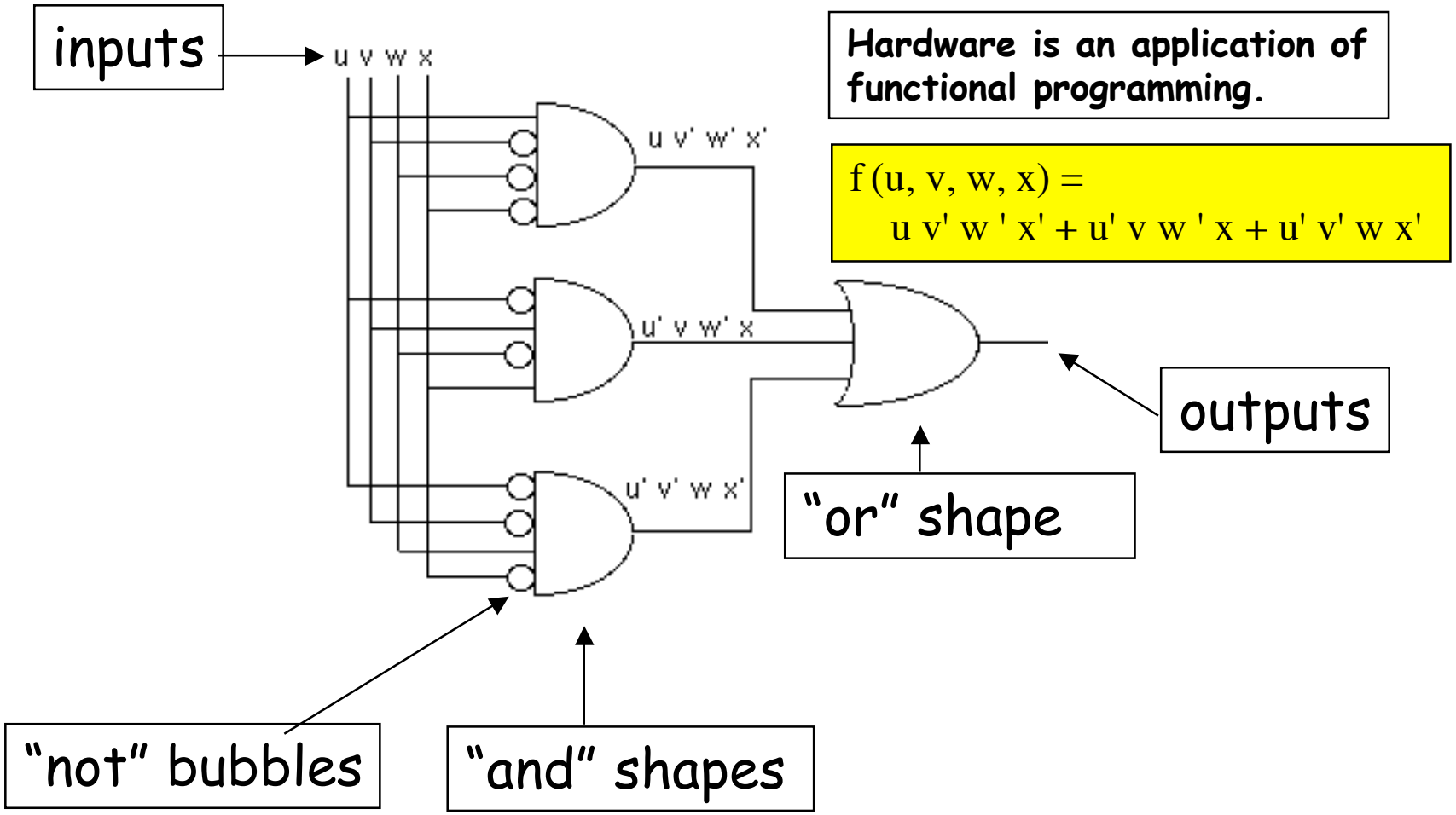
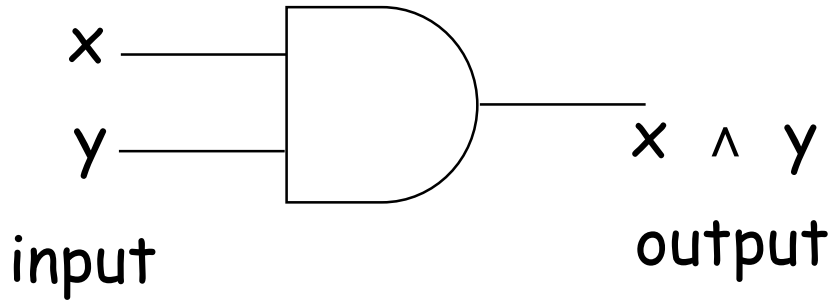
Logic Circuit Synthesis

Synthesizing Switching Functions

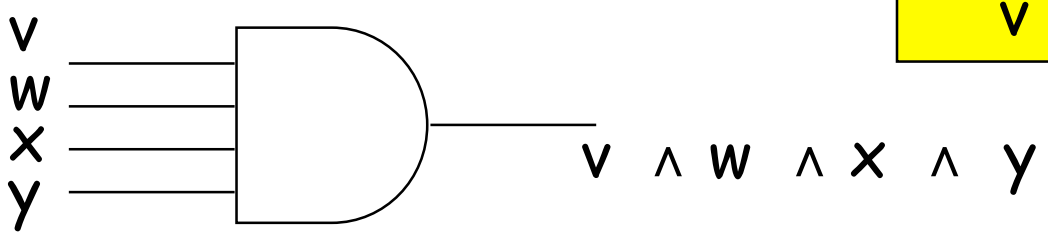
A "logic circuit" is composed of switching functions



"and" gates

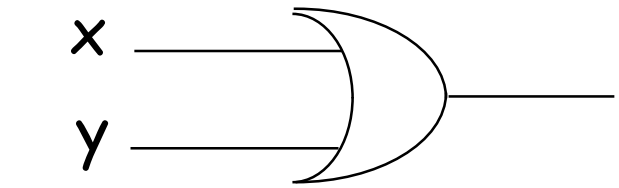


Engineering:
x y



Engineering:
v w x y

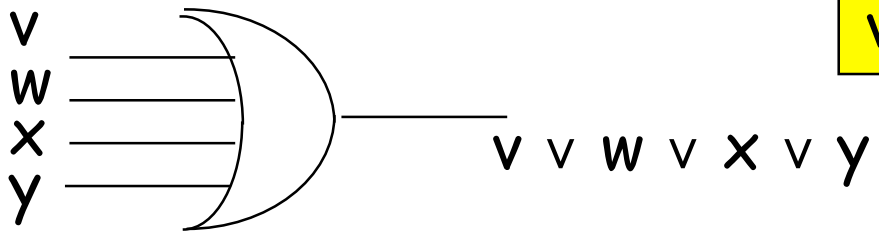
"or" gates



input

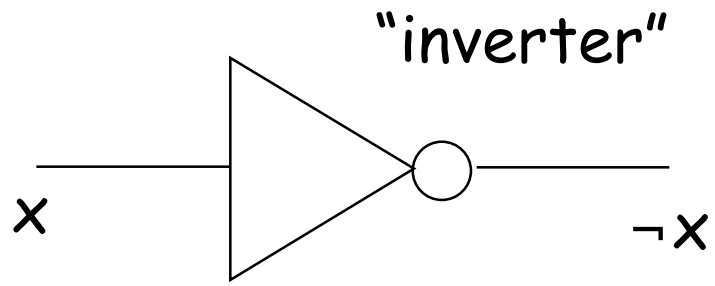
output

Engineering:
x + y

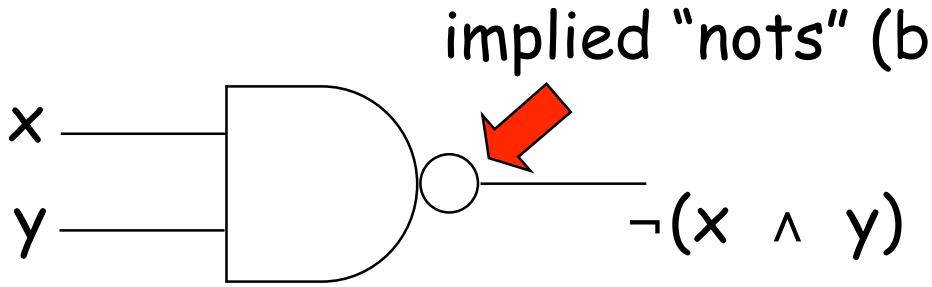


Engineering:
v + w + x + y

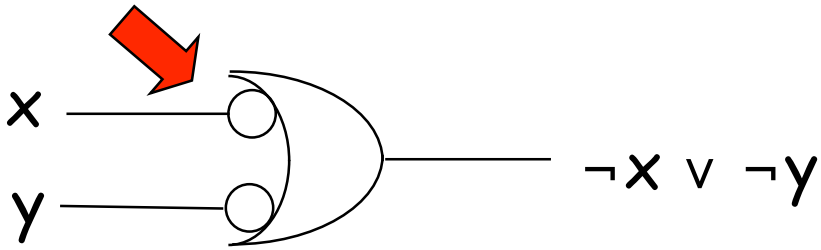
"not" Gates



x′



(x y)′



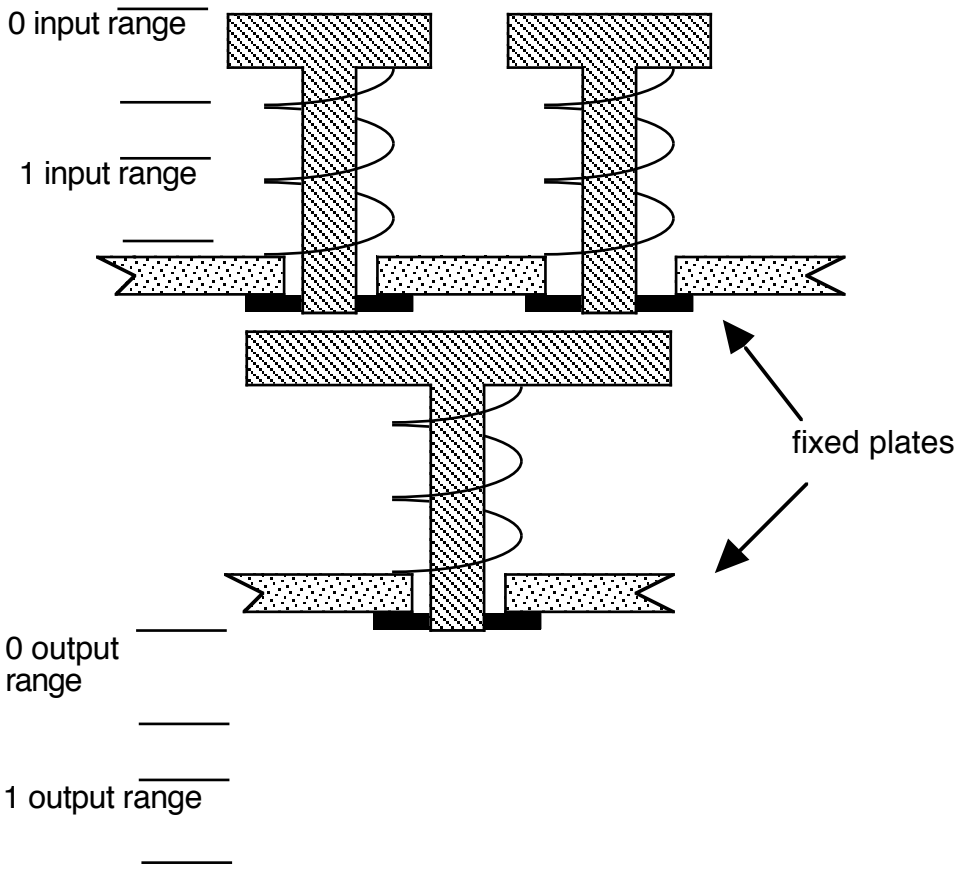
x′ + y′

Gates are Symbolic

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

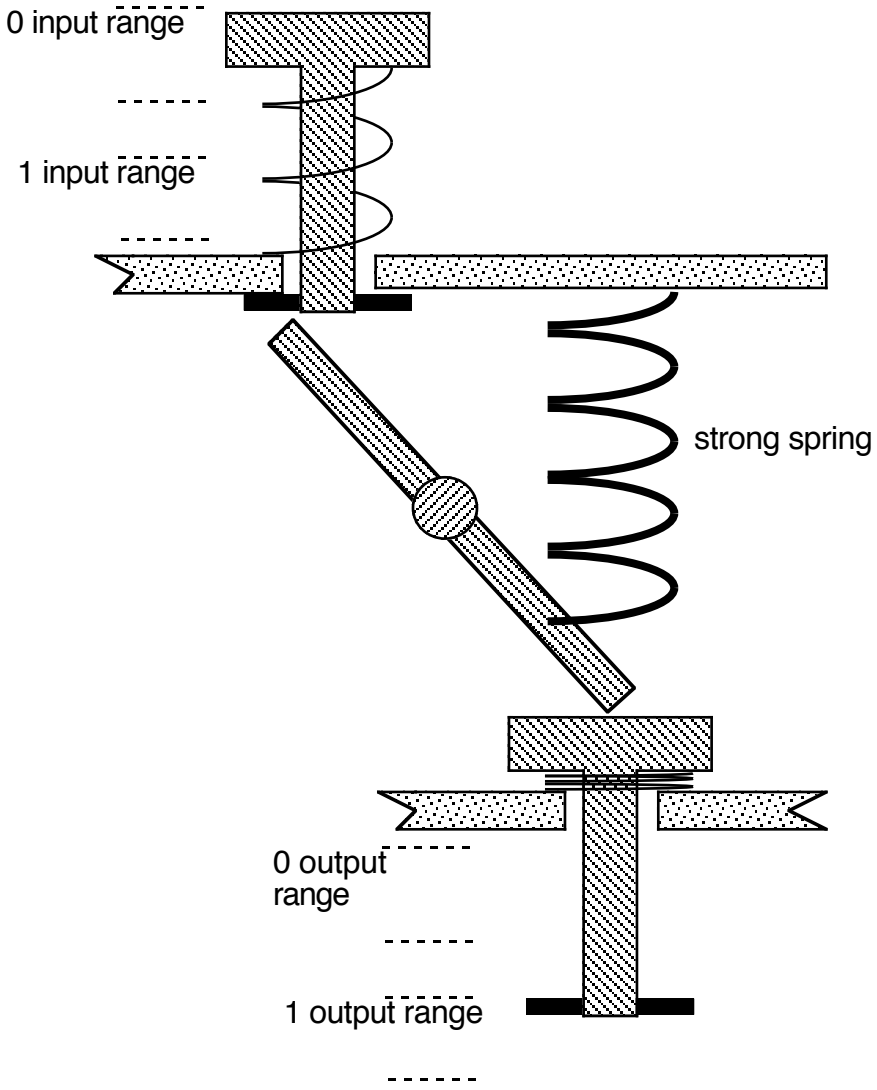
Mechanical Gates

"or" gate



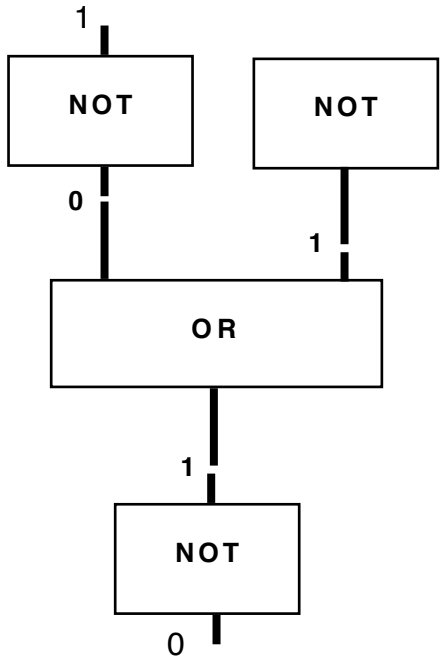
Mechanical Gates

"inverter"



Mechanical Gates

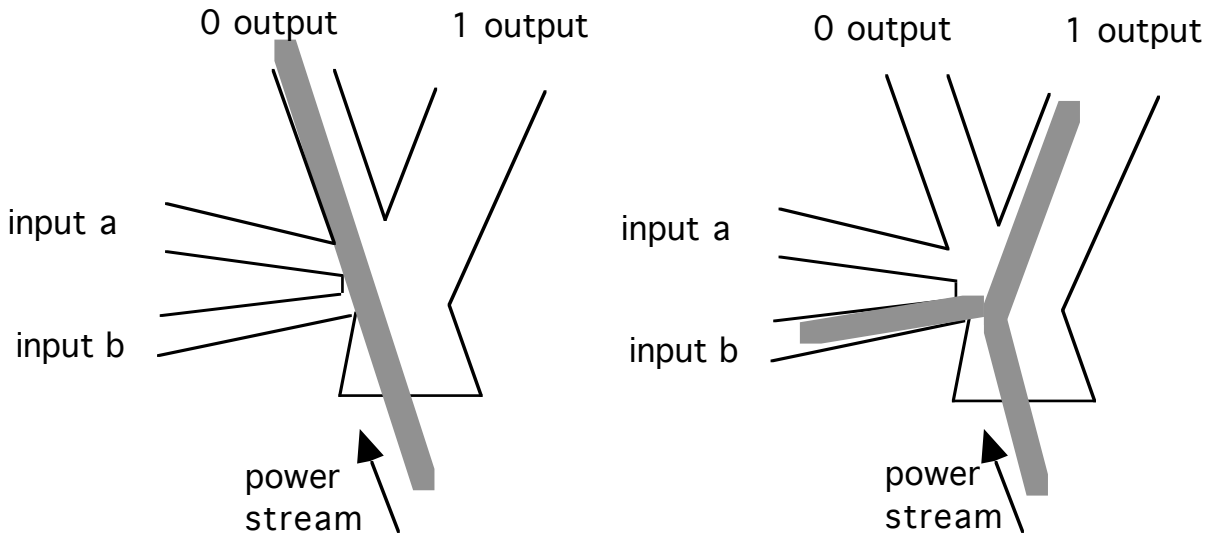
"and" gate, using DeMorgan's law



"Nanotechnology"

Fluid Gates

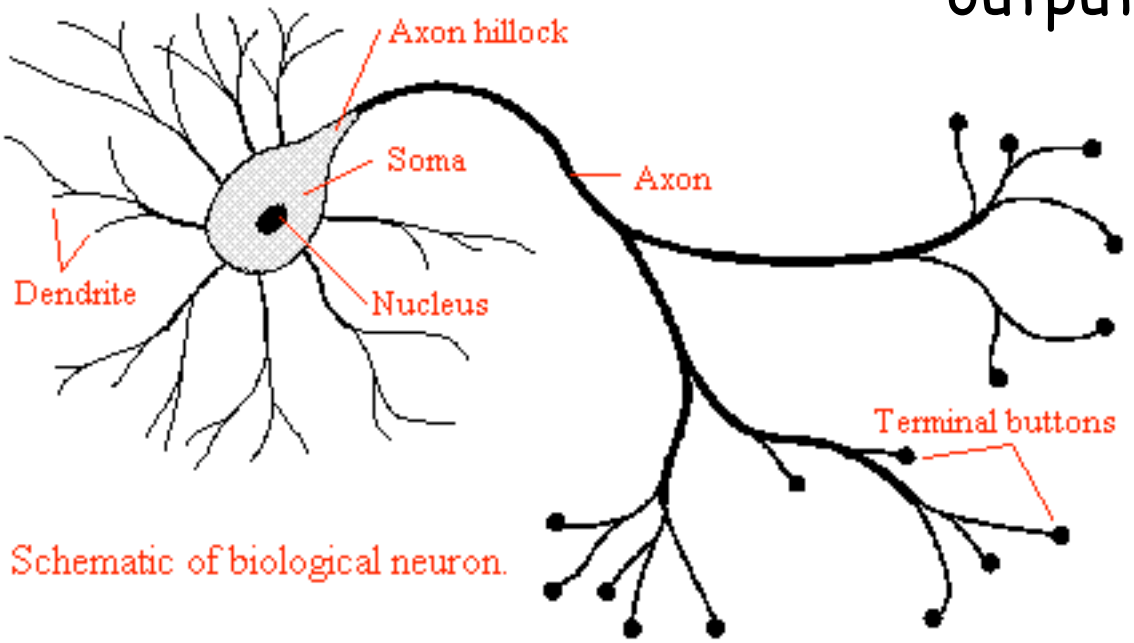
"or" gate



Biological Gates

input

output



Schematic of biological neuron.

DNA-based Gates

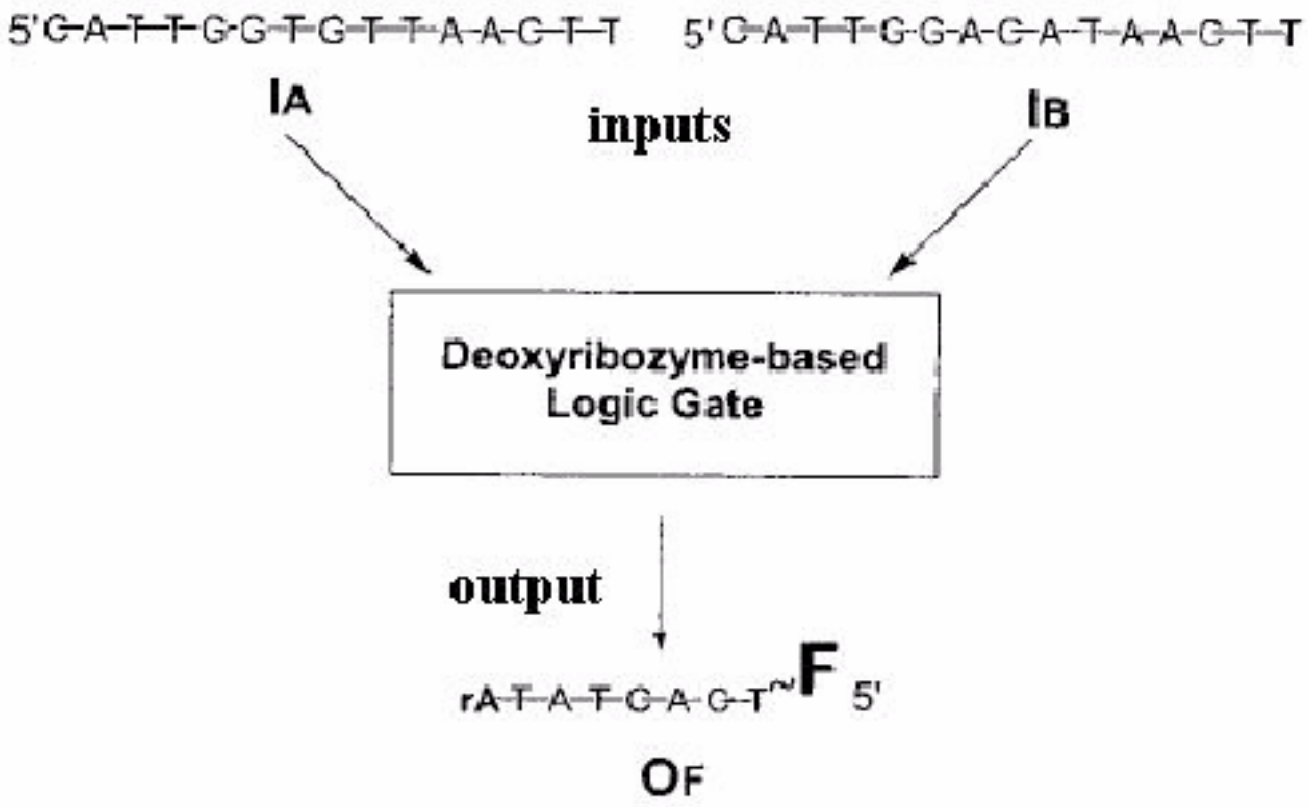
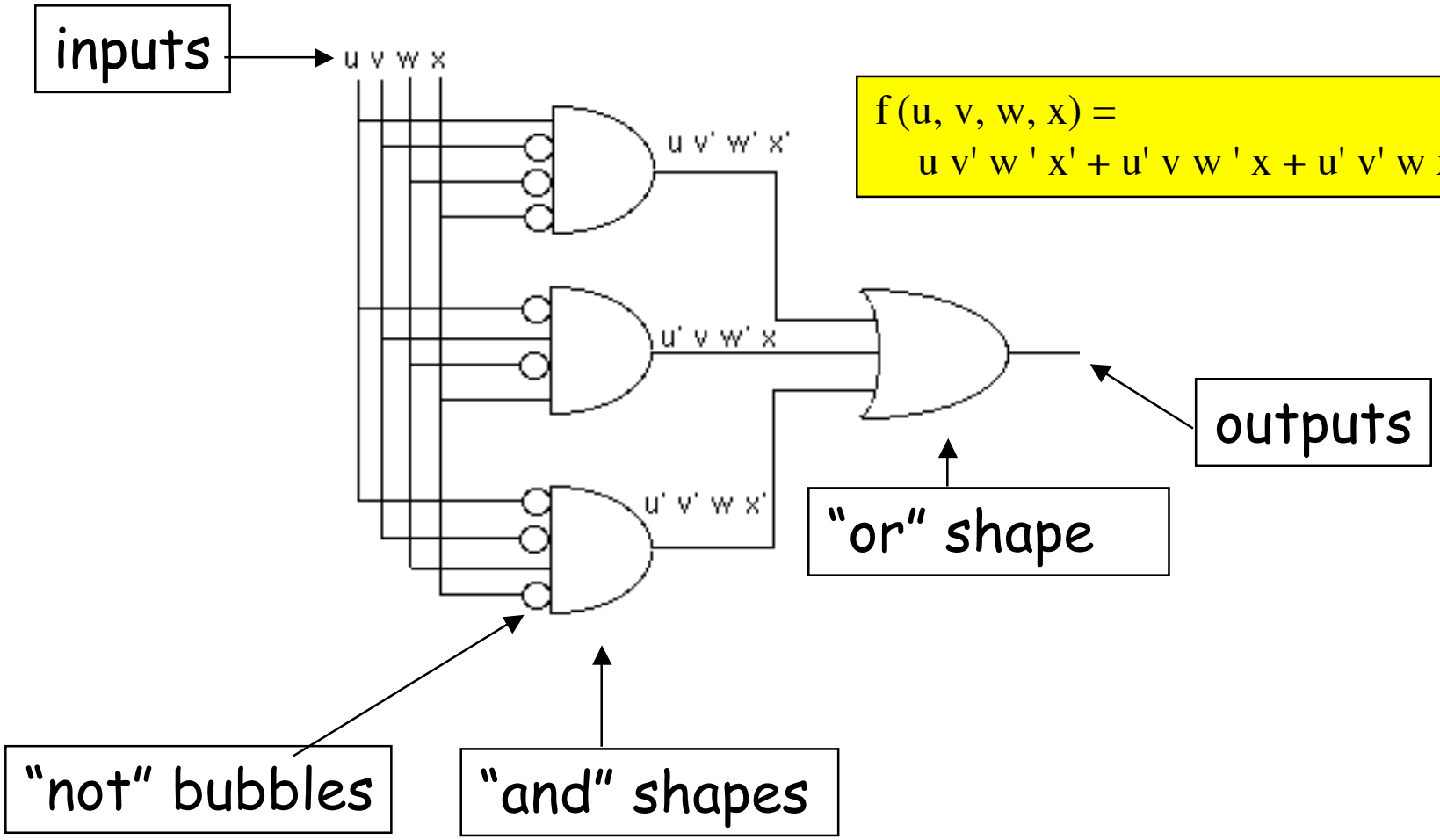


Figure 8: schematic of a logic deoxyribozyme-based logic gate with the input and output oligonucleotides.

Synthesizing Switching Functions

A "logic circuit" is composed of switching functions



Ways to Specify Switching Functions

- Logic circuit
- Functional expression
  - SOP (sum-of-products) form
    - Minterm form
    - Other forms
- 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

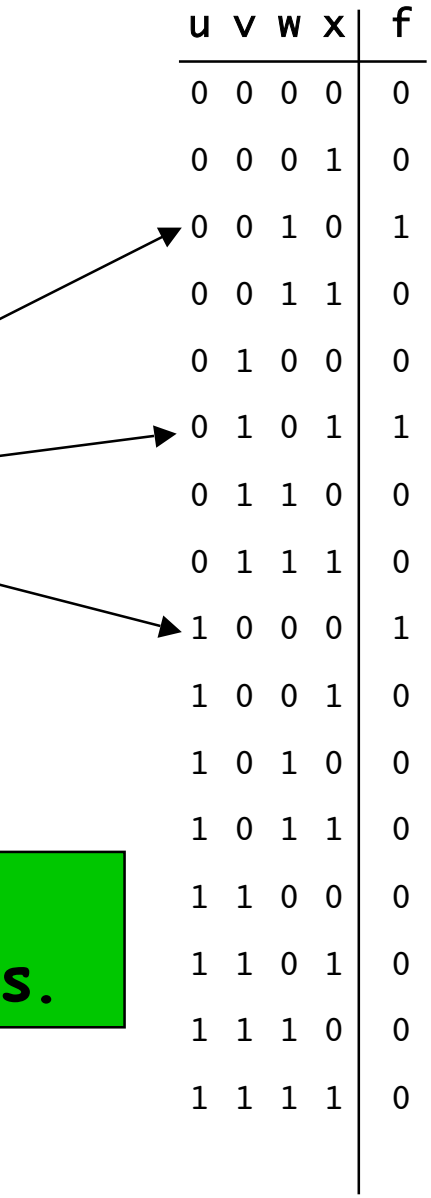
Note the Connection

f (u, v, w, x) =

u v' w ' x'

+ u' v w ' x

+ u' v' w x'



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

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

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 combinations of n variables.
- 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: $2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$

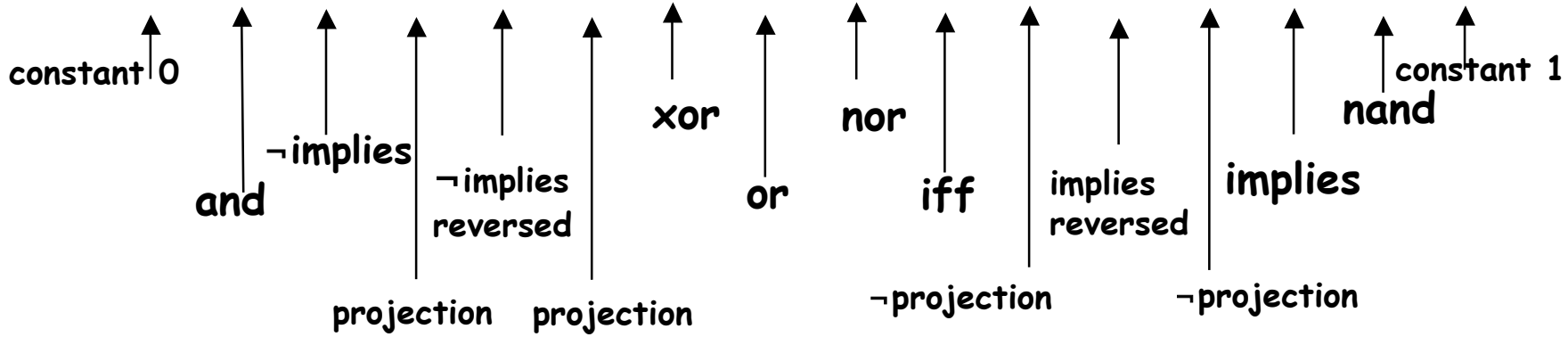
Each level **squares** the previous, since

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

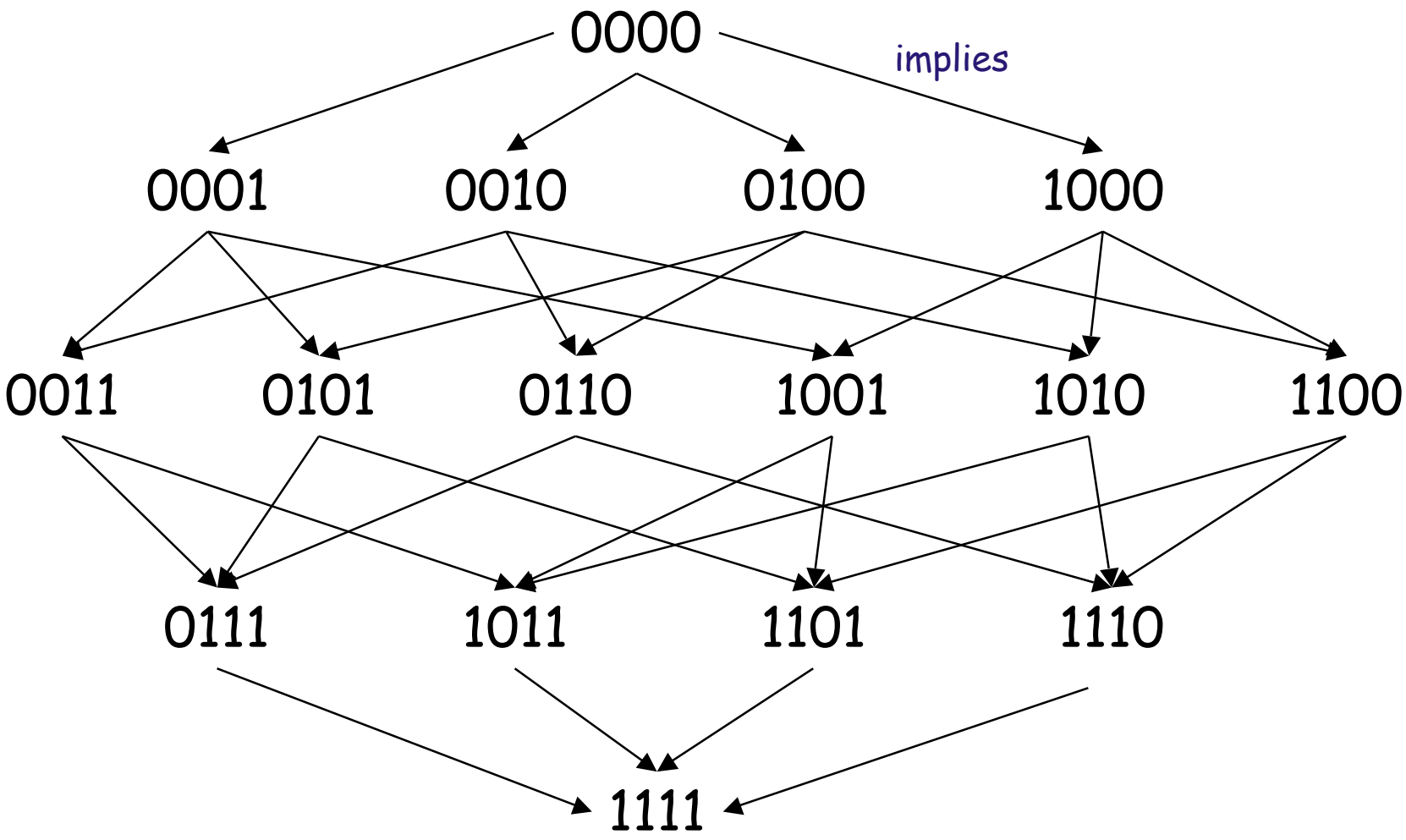
- Also remember that $2^{10} = 1024$ approx. 1000
  2^{20} = approx. 1,000,000 etc.

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



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 $\in$ {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 possible):

  $0 \rightarrow 00$

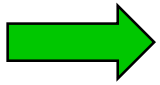
  $1 \rightarrow 01$

  $2 \rightarrow 10$

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 → 00
1 → 01
2 → 10

Here first argument becomes uv, second becomes wx.

Encoded Function

wx \ 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

wx \ 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.

wx \ uv	00	01	10
00	0	0	1
01	0	1	0
10	1	0	0

Call this f_2.

wx \ 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

wx \ uv	00	01	10
00	0	0	1
01	0	1	0
10	1	0	0

f_2

wx \ 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	?	?

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

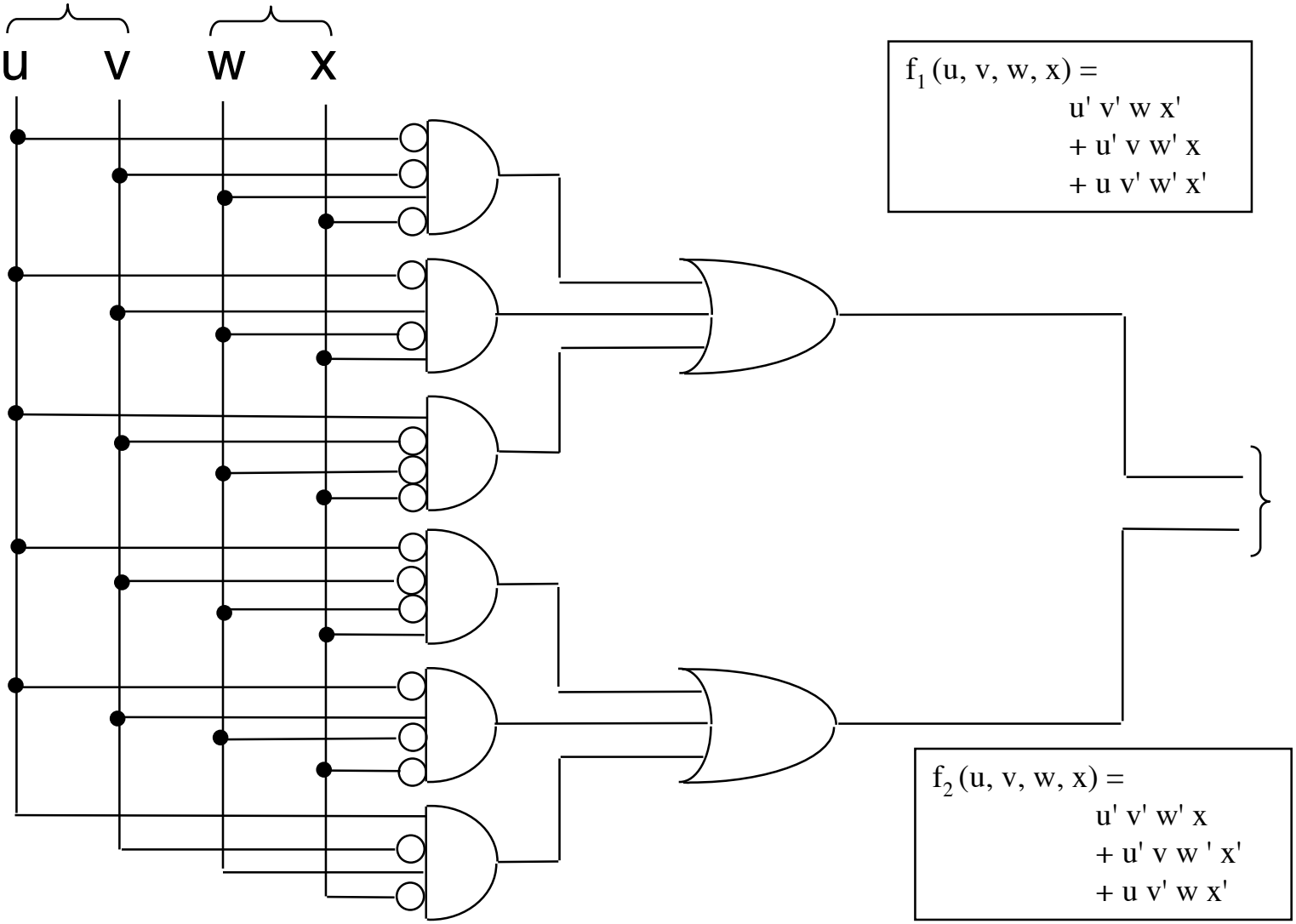
$u' v' w' x$

$+ u' v w' x'$

$+ u v' w x'$

Realize each function by gates

Mod 3 adder

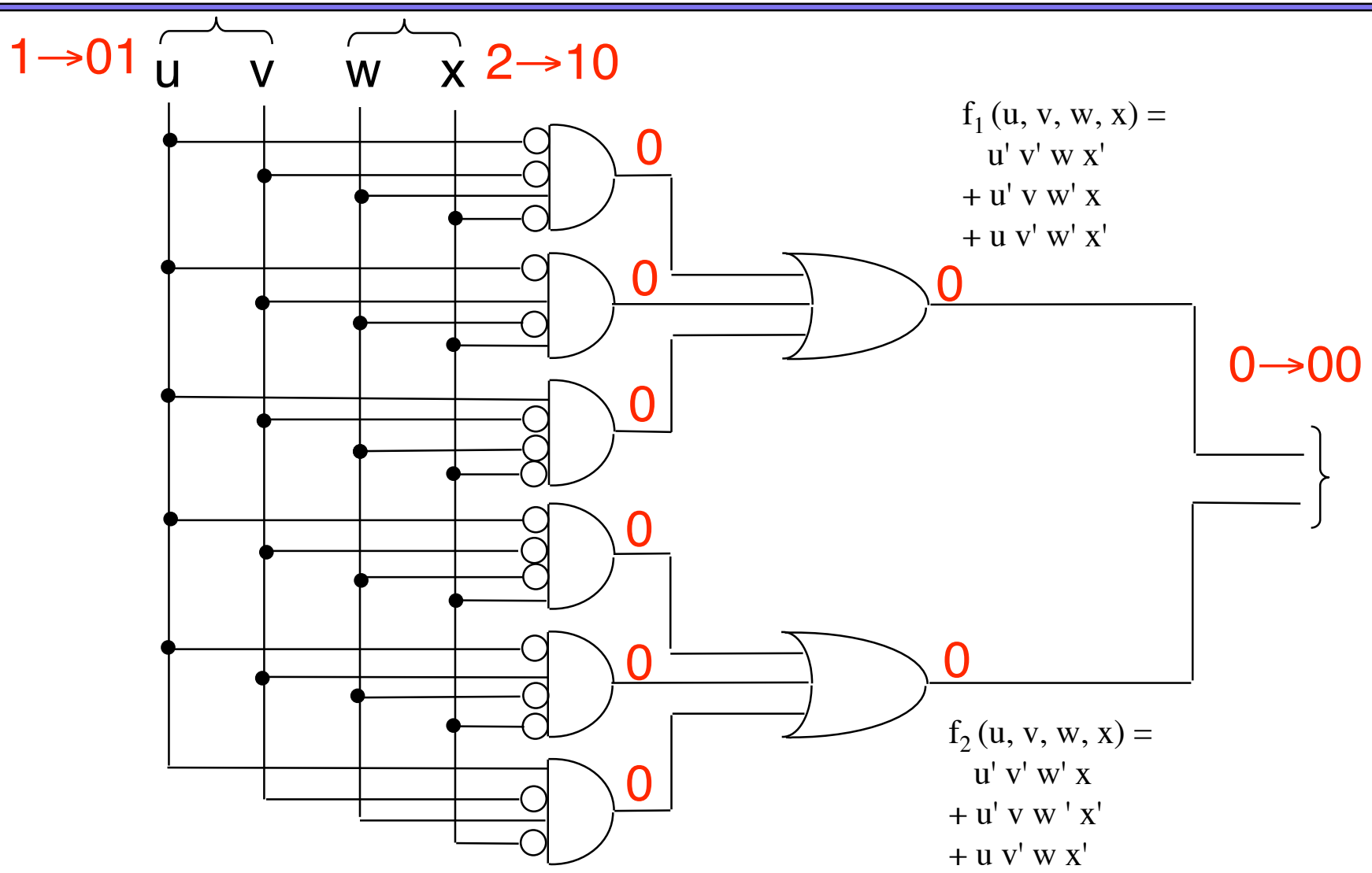


$f_1(u, v, w, x) =$
u' v' w x'
+ u' v w' x
+ u v' w' x'

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

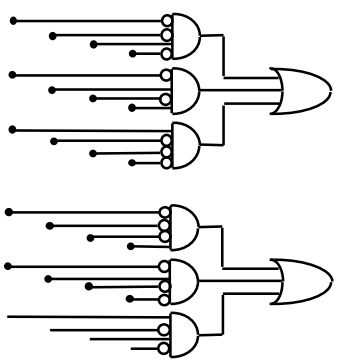
Check by "Reverse Engineering"

(Try all combinations; one combination is shown)



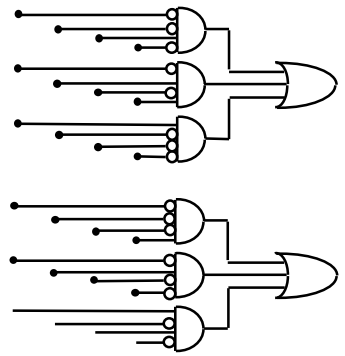
Rex Program for Checking all Combinations

```
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
    ];
```



Scheme Program for Checking all Combinations

```
; The test encoder/decoder

(define (encode d)
  (case d
    ('0 '(#f #f))
    ('1 '(#f #t))
    ('2 '(#t #f))
    (else (error "undefined encode" d))))

(define (decode e)
  (cond
    ((equal? e '(#f #f)) 0)
    ((equal? e '(#f #t)) 1)
    ((equal? e '(#t #f)) 2)
    (else ((error "undefined decode" e)))))

(define (add3 i j)
  (modulo (+ i j) 3))

(define (addByCircuit i j)
  (decode (circuit (encode i) (encode j))))
```



```
; The circuit simulation

(define (circuit y z)
  (let ((u (first y))
        (v (second y))
        (w (first z))
        (x (second z)))
    (list
     (or
       (and (not u) (not v)       w  (not x))
       (and (not u)       v  (not w)       x)
       (and       u  (not v) (not w) (not x)))
     (or
      (and (not u) (not v) (not w)       x)
      (and (not u)       v  (not w)  (not x))
      (and       u  (not v)       w   (not x))))))
```

Testing Every Combination

```
(test (addByCircuit 0 0) 0)
(test (addByCircuit 0 1) 1)
(test (addByCircuit 0 2) 2)

(test (addByCircuit 1 0) 1)
(test (addByCircuit 1 1) 2)
(test (addByCircuit 1 2) 0)

(test (addByCircuit 2 0) 2)
(test (addByCircuit 2 1) 0)
(test (addByCircuit 2 2) 1)
```

```
Test 1: (addByCircuit 0 0) succeeds with result: 0
Test 2: (addByCircuit 0 1) succeeds with result: 1
Test 3: (addByCircuit 0 2) succeeds with result: 2
Test 4: (addByCircuit 1 0) succeeds with result: 1
Test 5: (addByCircuit 1 1) succeeds with result: 2
Test 6: (addByCircuit 1 2) succeeds with result: 0
Test 7: (addByCircuit 2 0) succeeds with result: 2
Test 8: (addByCircuit 2 1) succeeds with result: 0
Test 9: (addByCircuit 2 2) succeeds with result: 1
```

Logic Simplification

- Consider the Boole/Shannon Expansion:

  $f(x, y) = x\, f(1, y) + x'\, f(0, y)$

- where y represents a bunch of variables.
- Without simplification, repeated expansion can be unwieldy.
- The following slides show ways to simplify.

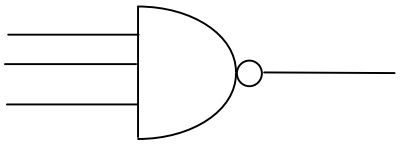
Some Logic Simplifications

$f(x, y) = x\,G(y) + x'\,H(y)$

- If G is the 0 function, then $f(x, y) = x'H(y)$.
- If H is the 0 function, then $f(x, y) = xG(y)$.
- If $G = H$ (as functions), then $f(x, y) = G(y)$.
- If G is the 1 function, then $f(x, y) = x + H(y)$ [no x on right].
- If H is the 1 function, then $f(x, y) = x' + G(y)$ [no x on right].

Bubble Logic

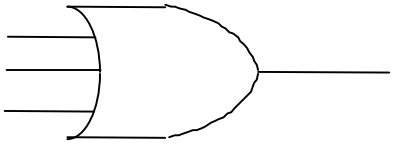
nand



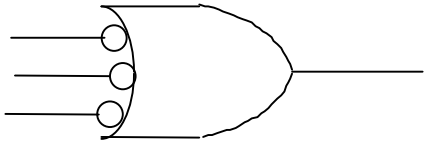
DeMorgan:

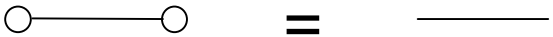
nand(a, b) =
and(a, b)' =
or(a', b')

or



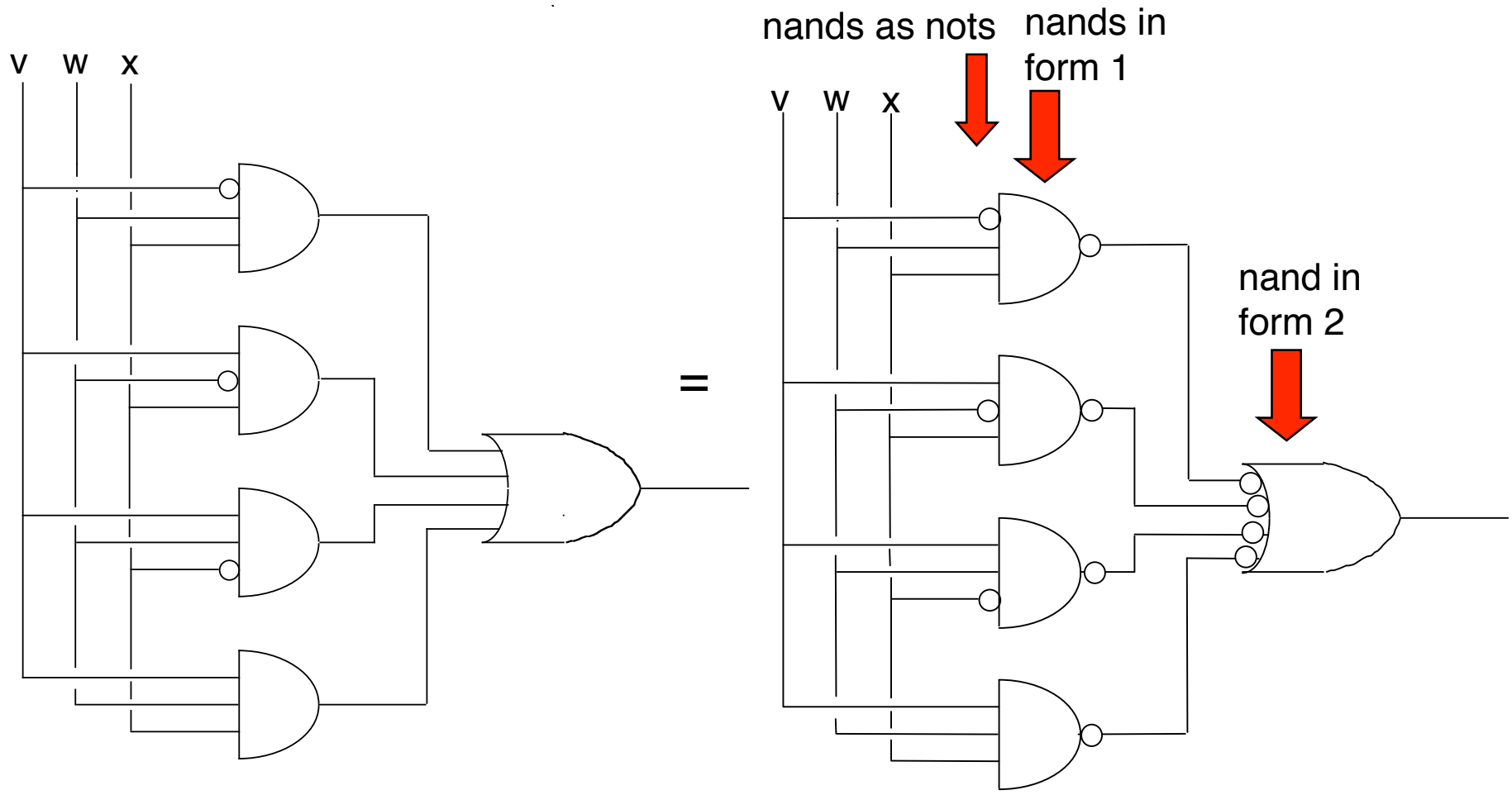
nand, another way





(a')' = a

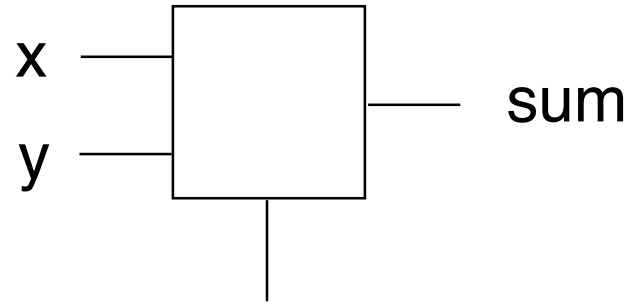
SOP form using nand's only



Common Logic Packages

- mod-2 adder
- mod-2 adder with carry-in
- 4-bit adder
- decoder (binary to one-hot)
- encoder (one-hot to binary)
- (MUX) multiplexer
- (DMUX) demultiplexer

mod-2 adder with carry-out

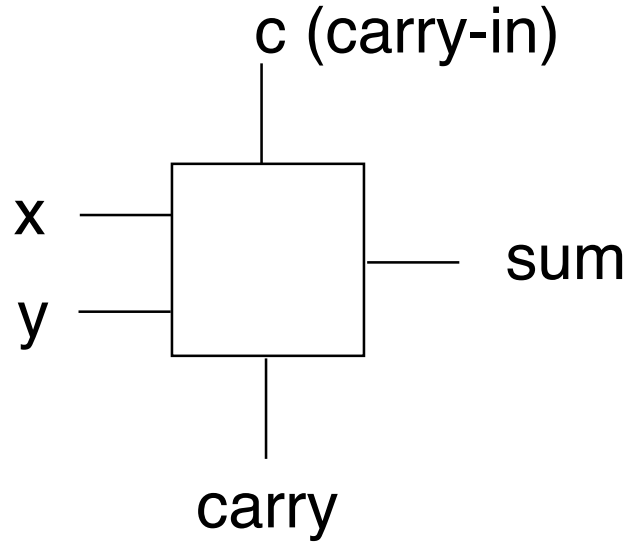


exclusive or

$sum(x, y) = x \oplus y = xy' + x'y$

$carry(x, y) = x\ y$

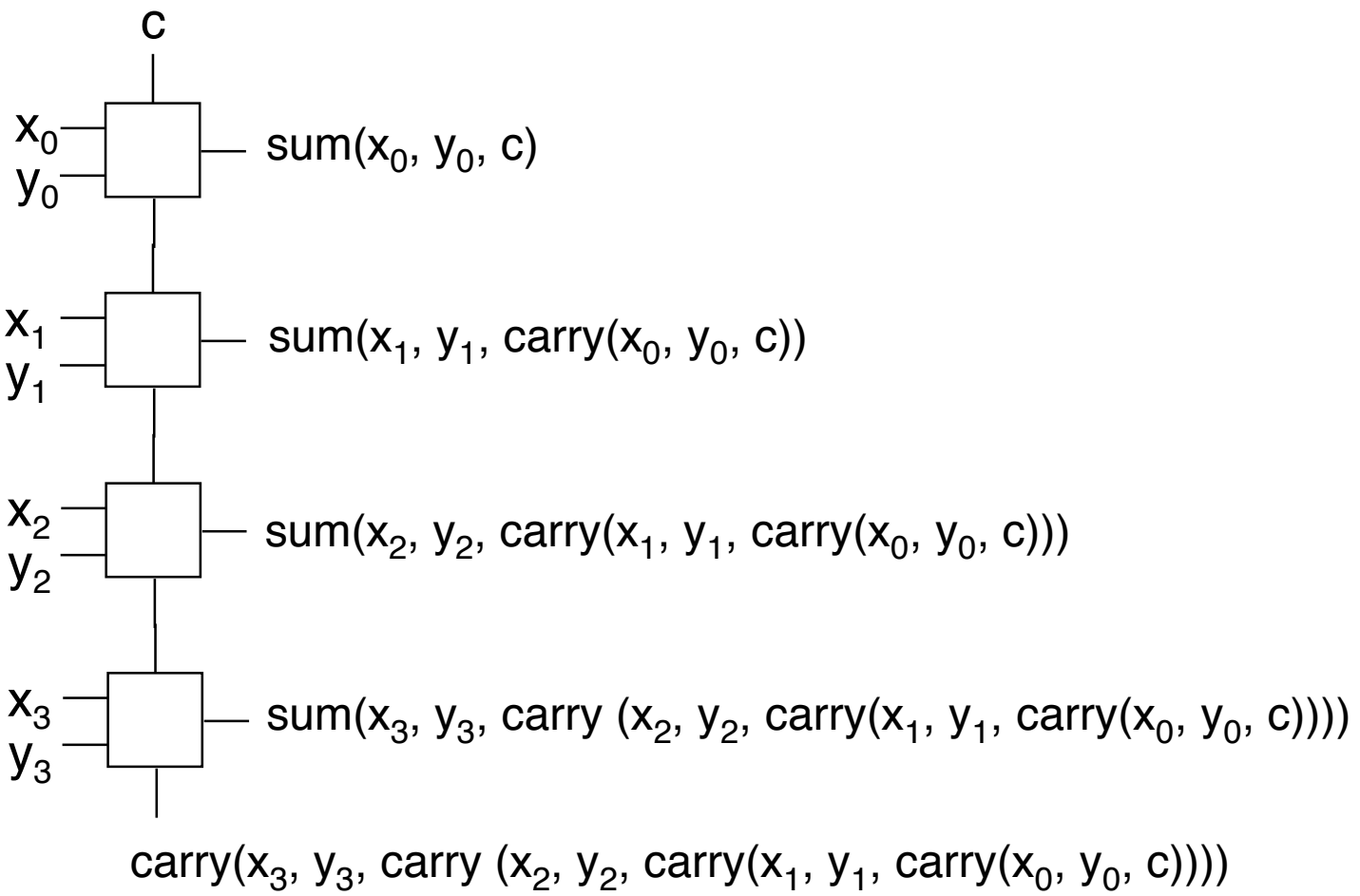
mod-2 adder with carry-in/out



$$\text{sum}(x, y, c) = x \oplus y \oplus c$$

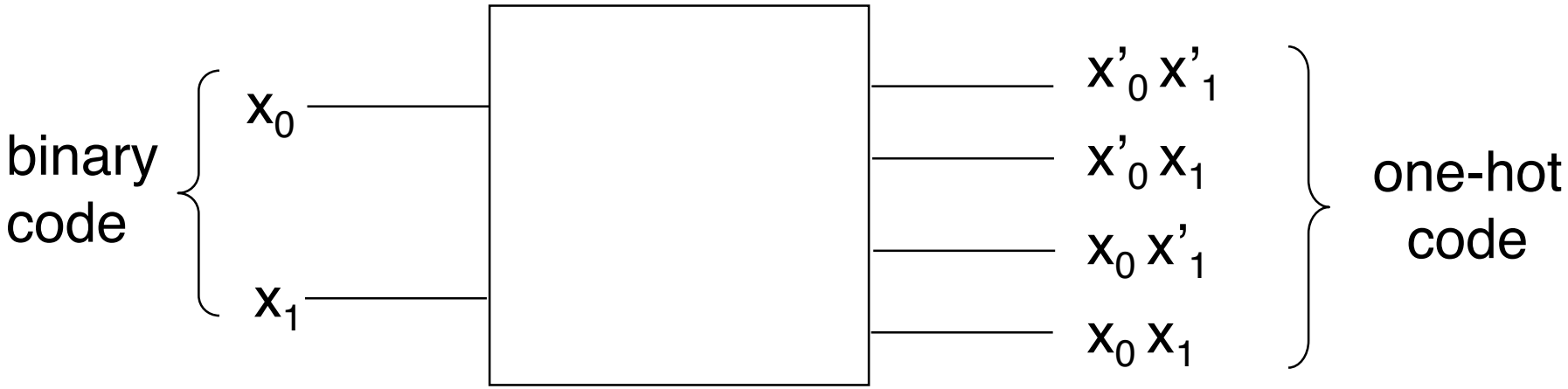
$$\text{carry}(x, y, c) = x\,y + x\,c + y\,c = \text{majority}(x, y, c)$$

4-bit ripple-carry adder

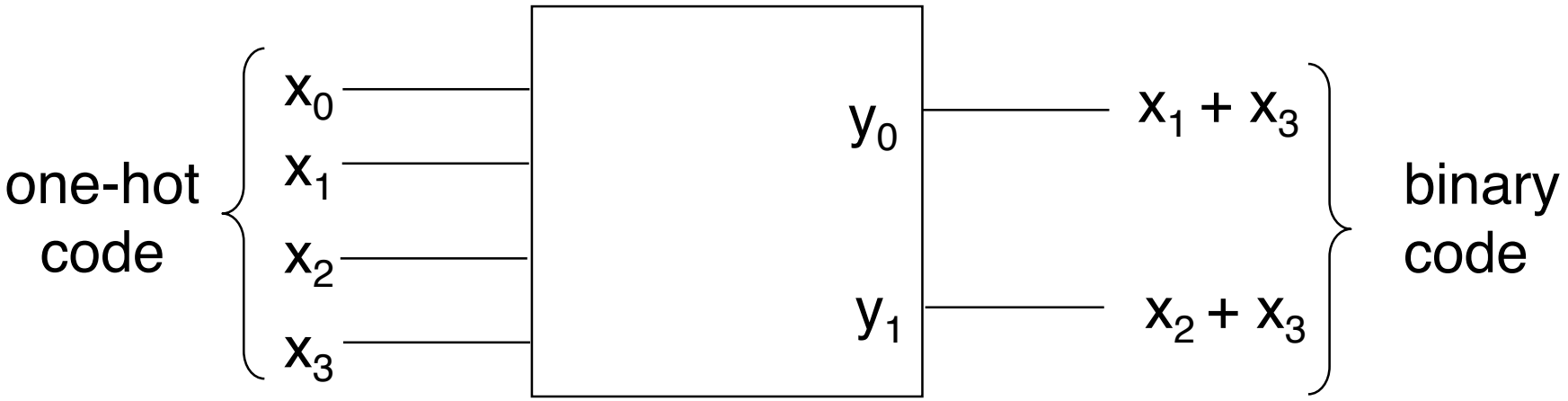


Note: There are other ways to implement this.

4-bit decoder



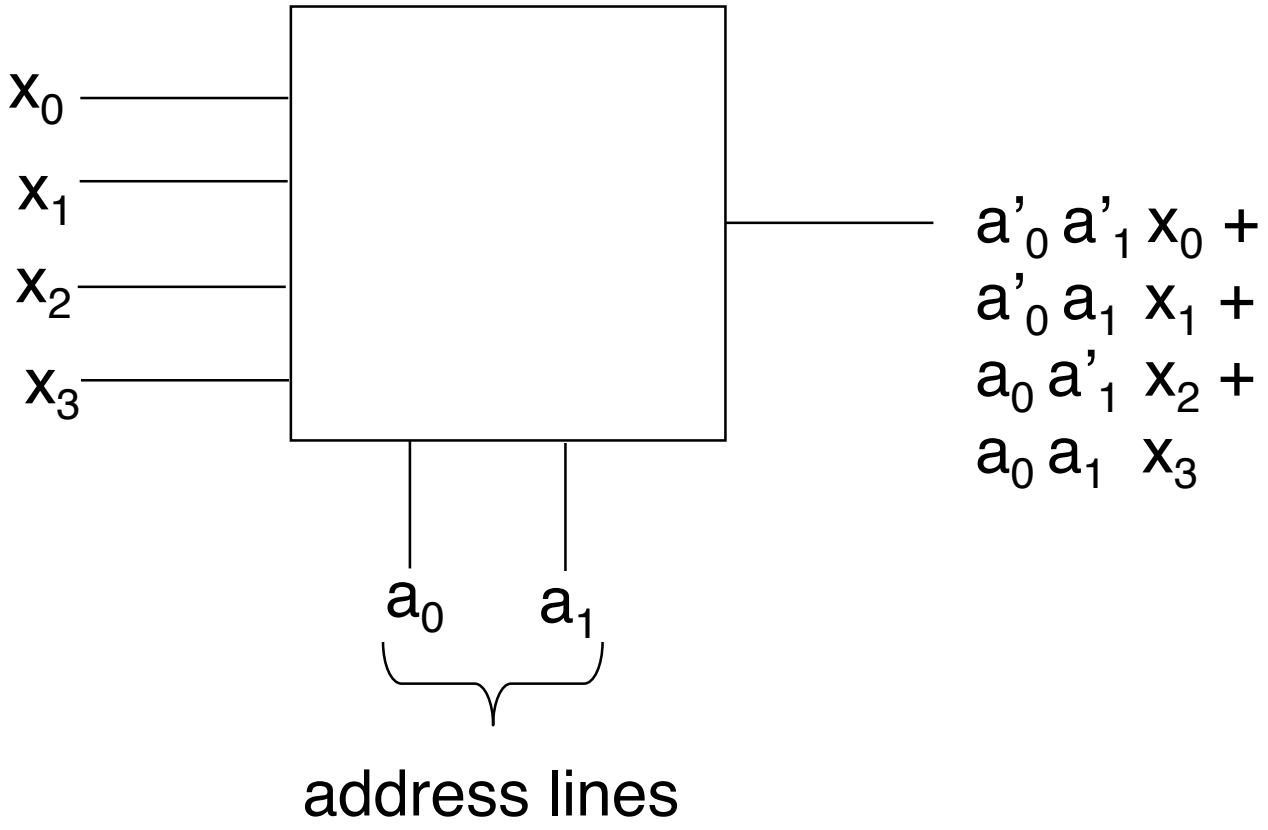
4-bit encoder



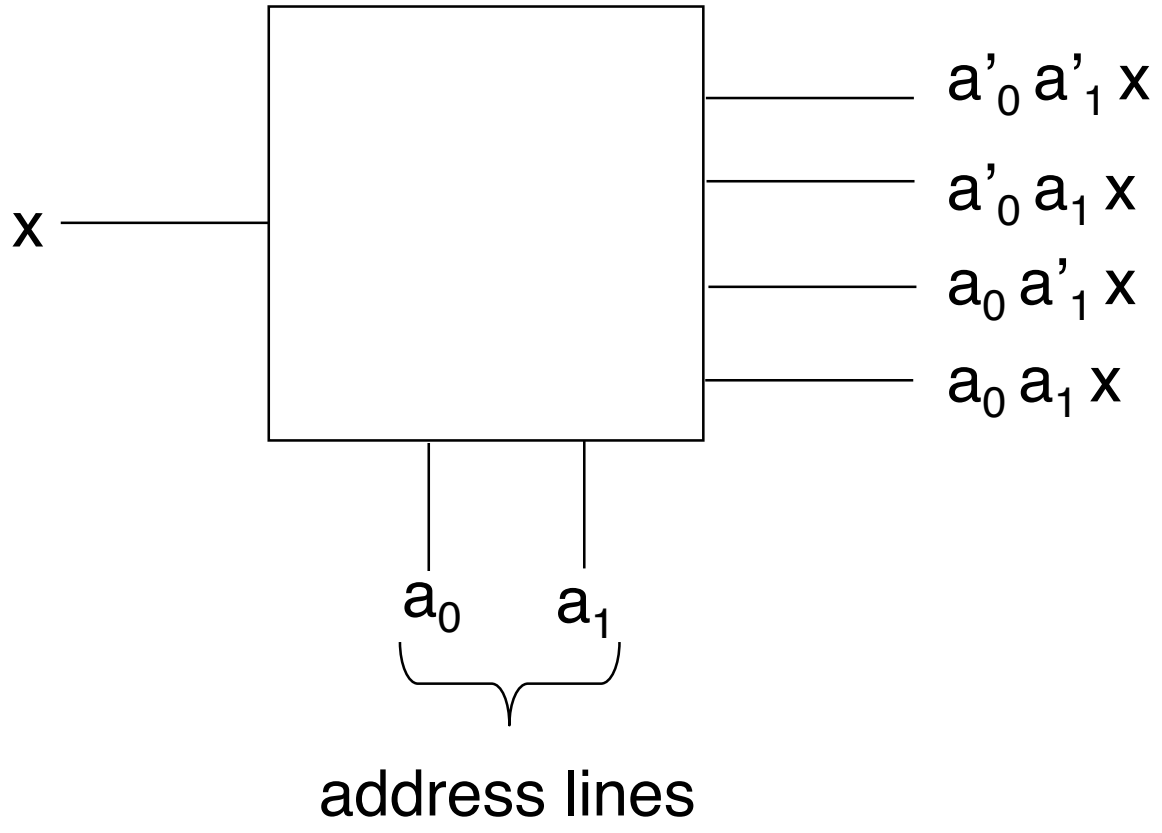
x_0	x_1	x_2	x_3	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

unlisted combinations are assumed not to occur

multiplexer



demultiplexer



Exercises

- How would you build a decoder out of a demultiplexor?
- How would you build a 16-way multiplexor out of 4-way multiplexors?
- How would you build a 16-way demultiplexor out of 4-way demultiplexors?

Logic Rebuses