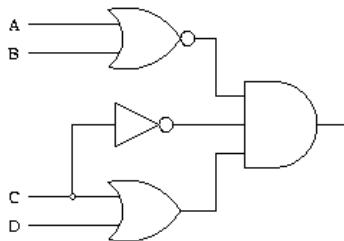


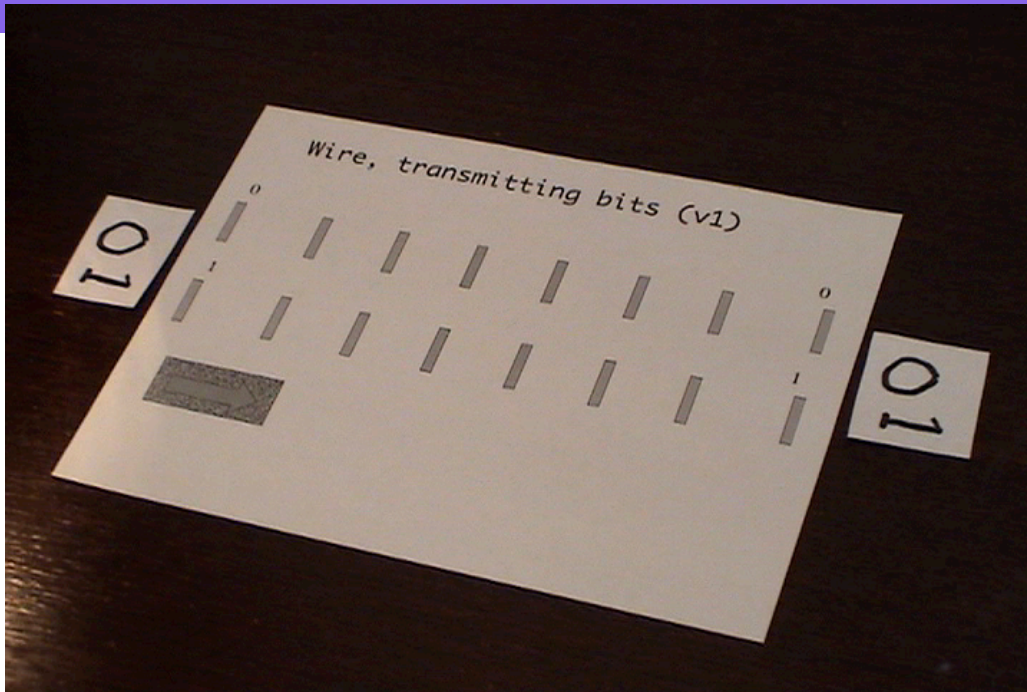
Lecture 3: Gates

CS442: Great Insights in Computer Science
Michael L. Littman, Spring 2006

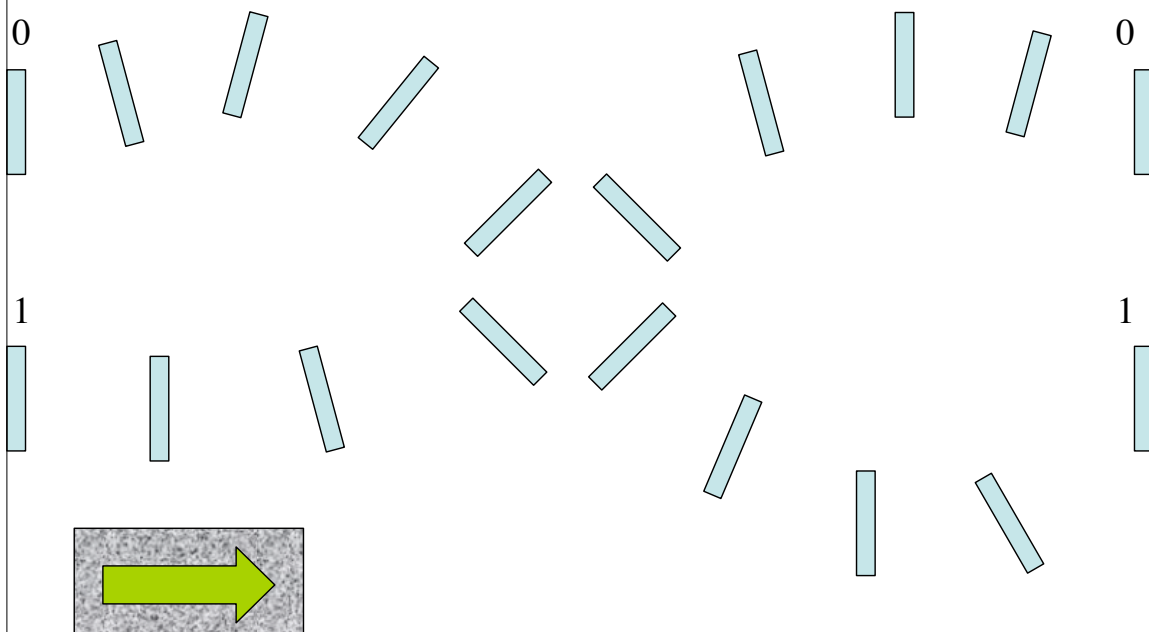
Name These Gates

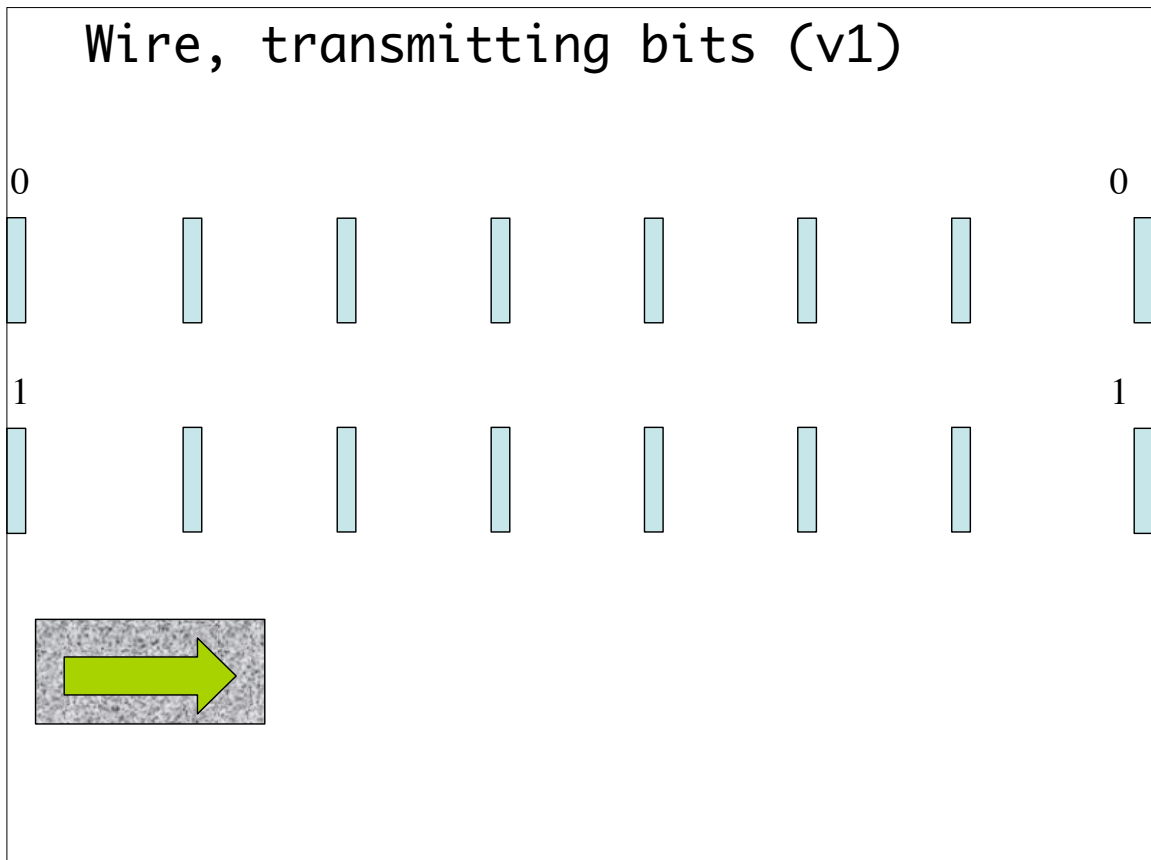
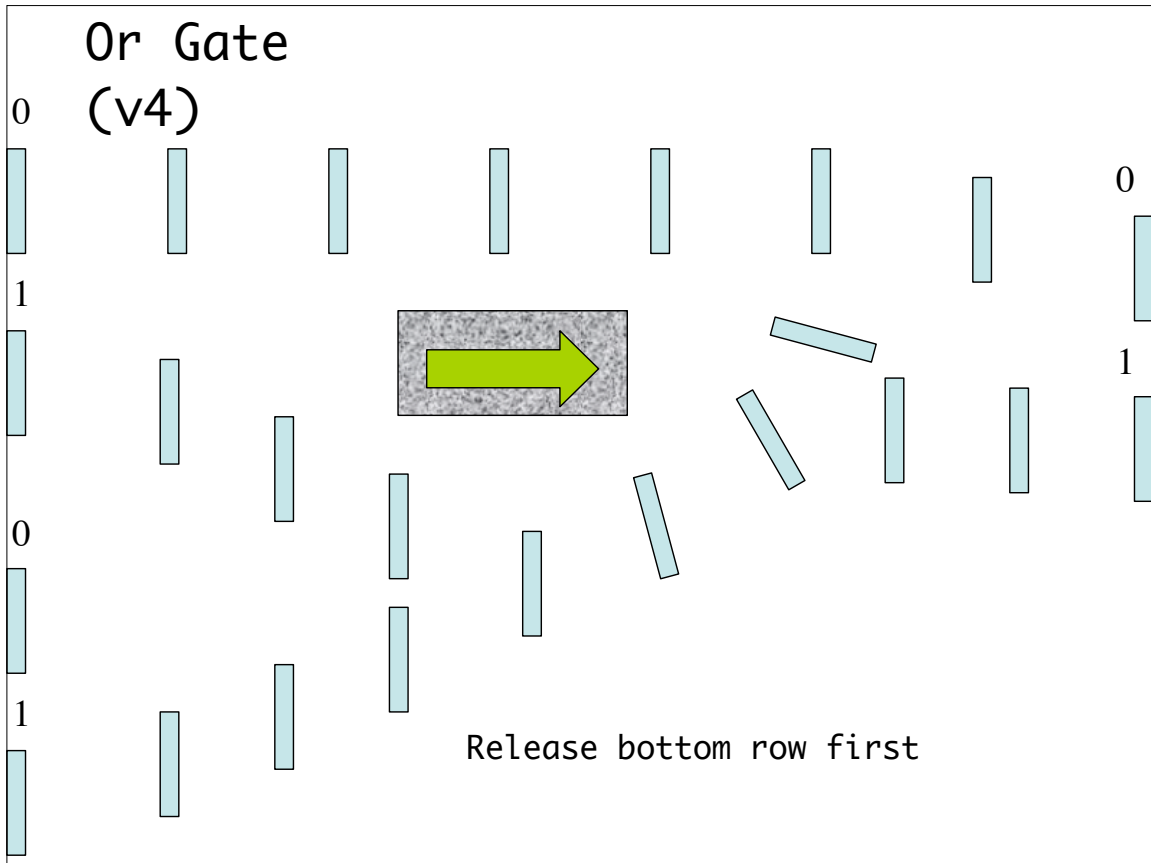


Movie



NOT Gate (v3)

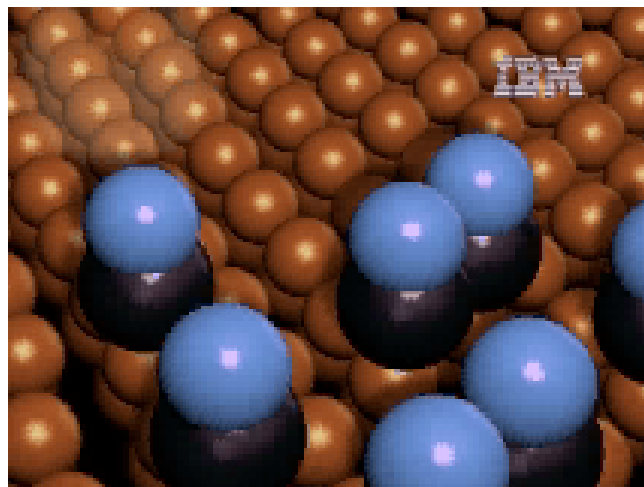




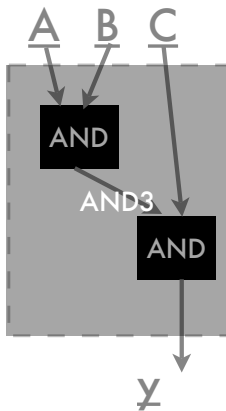
Could It Work?

- My domino “or” gate requires 24 dominoes.
- The first Pentium processor had 3.3M transistors, or roughly 800K gates.
- So, perhaps 19M dominoes needed.
- World record for domino toppling: 4M.
- Oh, and the Pentium did its computations 60M times a second, whereas dominoes might require a week to set up *once*.

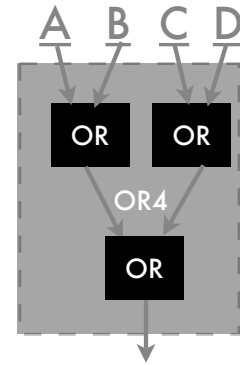
Atomic Dominoes: IBM



A Few New Gates

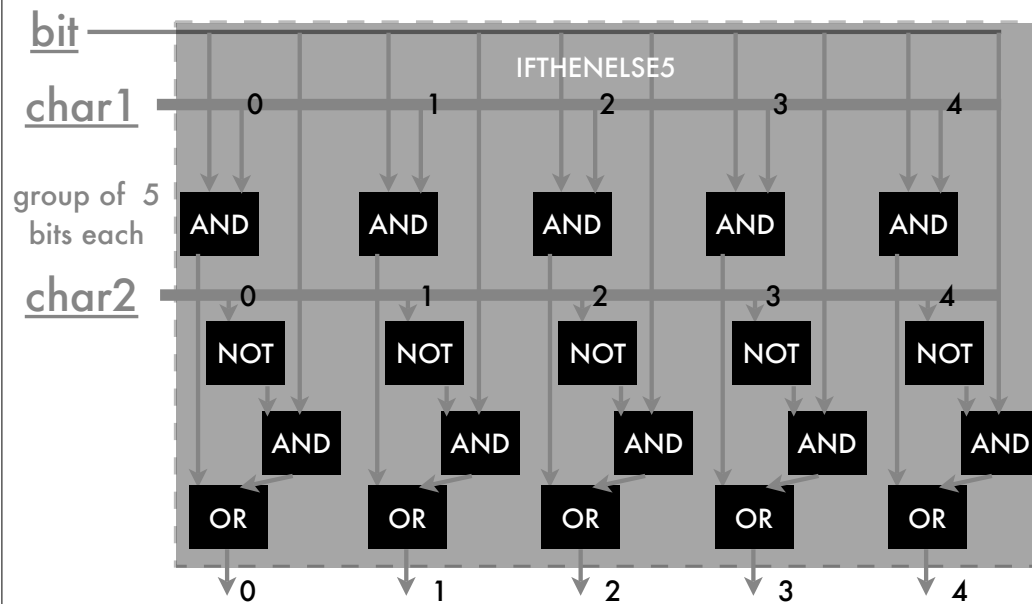


```
def AND3(A, B, C):
    x = A and B
    y = x and C
    return y
```



```
def OR4(A, B, C, D):
    return (A or B) or (C or D)
```

IFTHENELSE5



Textual Version

```
def IFTHENELSE5(bit, char1, char2):  
    return [(char1[0] and bit) or (char2[0] and not bit),  
            (char1[1] and bit) or (char2[1] and not bit),  
            (char1[2] and bit) or (char2[2] and not bit),  
            (char1[3] and bit) or (char2[3] and not bit),  
            (char1[4] and bit) or (char2[4] and not bit)]
```

- Takes 11 bits as input and makes 5 as output. For clarity, the bits are grouped.
- `char1[0]` means the leftmost bit of the group called “char1”.
- “bit” selects char1 (True) or char2 (False).

Why “Or”, “And”, “Not”?

- In addition to being familiar, these gates are “universal”. That is, all other logical functions can be expressed using these building blocks.
- How many distinct logic functions on 2 bits?

Some Truth Tables

A	B	C	A	B	C	A	B	C	A	B	C
False	False	False	False	False	False	False	False	False	False	False	False
False	True	False	False	True	False	False	True	False	False	True	False
True	False	False	True	False	False	True	False	True	True	False	True
True	True	False	True	True	True	True	True	False	True	True	True

A	B	C	A	B	C	A	B	C	A	B	C
False	False	False	False	False	False	False	False	False	False	False	False
False	True	True	False	True	True	False	True	True	False	True	True
True	False	False	True	False	False	True	False	True	True	False	True
True	True	False	True	True	True	True	True	False	True	True	True

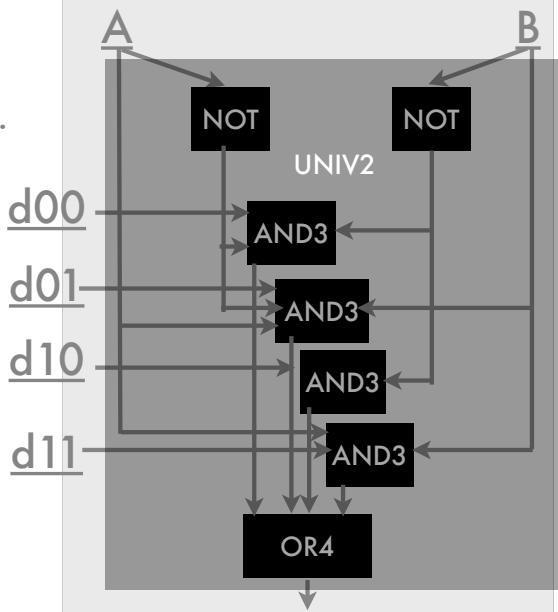
More Truth Tables

A	B	C	A	B	C	A	B	C	A	B	C
False	False	True	False	False	True	False	False	True	False	False	True
False	True	False	False	True	False	False	True	False	False	True	False
True	False	False	True	False	False	True	False	True	True	False	True
True	True	False	True	True	True	True	True	False	True	True	True

A	B	C	A	B	C	A	B	C	A	B	C
False	False	True	False	False	True	False	False	True	False	False	True
False	True	True	False	True	True	False	True	True	False	True	True
True	False	False	True	False	False	True	False	True	True	False	True
True	True	False	True	True	True	True	True	False	True	True	True

Universal Gate

- Take two inputs, A and B.
- Take four more inputs defining what the gate should output for each combination of A and B.
- Output the right bit!



Textual Version

```
def UNIV2(A, B, d00, d01, d10, d11):  
    w = AND3(not A, not B, d00)  
    x = AND3(not A, B, d01)  
    y = AND3(A, not B, d10)  
    z = AND3(A, B, d11)  
    return OR4(w,x,y,z)
```

Counting Boolean Functions

- With 2 input bits, there are $2^2=4$ rows of the truth table (combinations of truth assignments to these variables).
- Each row can take an output of true or false, for a total of $2^4=16$ tables.
- For n inputs: 2^{2^n} .

n	2^{2^n}
0	1
1	4
2	16
3	256
4	65536
5	4294967296
6	18446744073709551616
7	34028236692093846346337

Can Represent Them All

- Almost all multi-input functions require an enormous number of logic gates.
- However, the most useful ones can be represented succinctly.

Exception Checker

- The rule: i before e except after c.
- Exceptions:
 - cie
 - ?ei, where ? is not a c
- Goal: Given an encoding of a 7-letter word, output a bit that encodes whether it is an exception to this rule (True) or not (False).

5-bit Letter Codes

A	code
_	00000
a	00001
b	00010
c	00011
d	00100
e	00101
f	00110
g	00111

A	code
h	01000
i	01001
j	01010
k	01011
l	01100
m	01101
n	01110
o	01111

A	code
p	10000
q	10001
r	10010
s	10011
t	10100
u	10101
v	10110
w	10111

A	code
x	11000
y	11001
z	11010

Top-Down Design

```
# word is 7 groups of 5 bits each
# [False, False, False, False, False] is '_'
def exception(word):
    ps = exception3([False, False, False, False, False], word[0], word[1])
    p0 = exception3(word[0], word[1], word[2])
    p1 = exception3(word[1], word[2], word[3])
    p2 = exception3(word[2], word[3], word[4])
    p3 = exception3(word[3], word[4], word[5])
    p4 = exception3(word[4], word[5], word[6])
    return ((ps or p0) or p1) or (p2 or (p3 or p4))
```

Local Check

```
# [False, False, False, True, True] is 'c'
# [False, True, False, False, True] is 'i'
# [False, False, True, False, True] is 'e'
def exception3(x,y,z):
    ex1 = AND3(equal5(x,[False, False, False, True, True]),
               equal5(y,[False, True, False, False, True]),
               equal5(z,[False, False, True, False, True]))
    ex2 = AND3(not equal5(x, [False, False, False, True, True]),
               equal5(y,[False, False, True, False, True]),
               equal5(z, [False, True, False, False, True]))
    return ex1 or ex2
```

Equality Check

```
def equal5(char1, char2):  
    return (equal(char1[0],char2[0])  
            and (equal(char1[1],char2[1])  
                  and (equal(char1[2],char2[2])  
                          and (equal(char1[3],char2[3])  
                                  and equal(char1[4],char2[4]))))))  
def equal(bit1, bit2):  
    return ((bit1 and bit2) or (not bit1 and not bit2))
```

Next Time

- Read Hillis, Chapter 2.