Discussion Section 3

Sean Huang

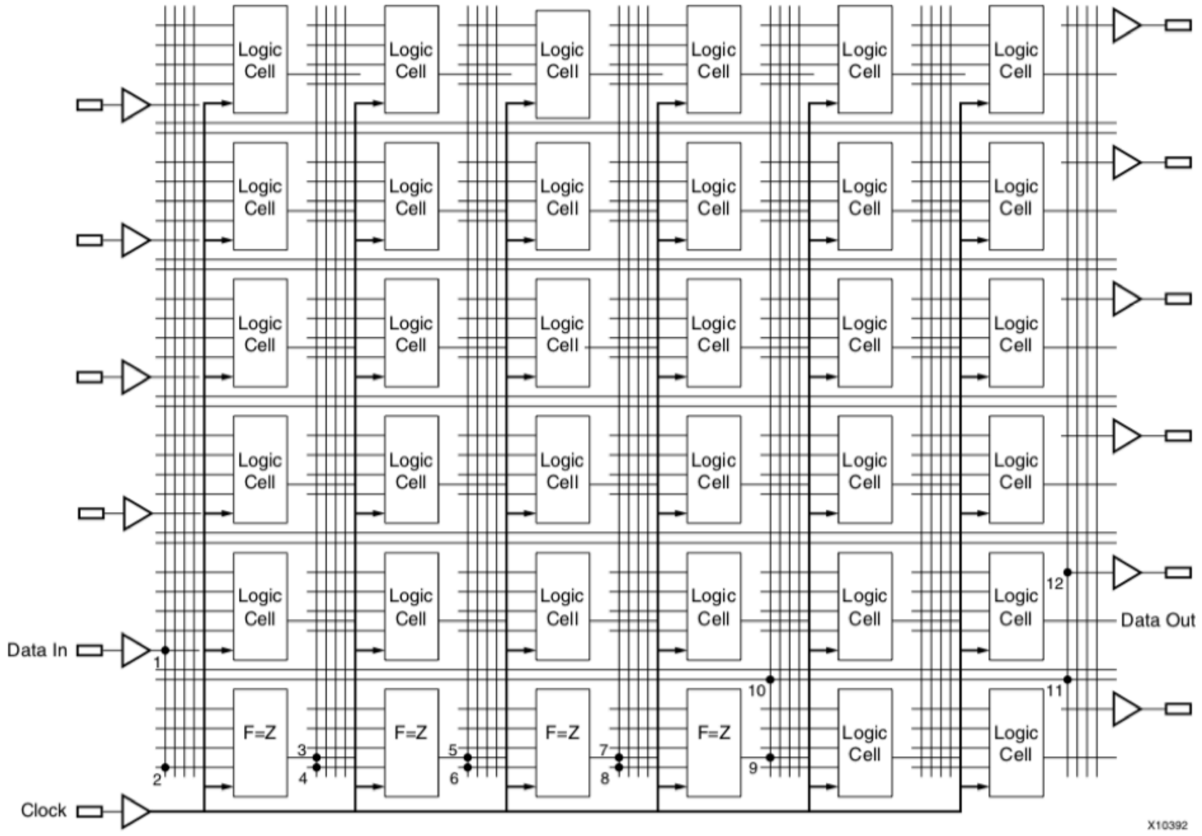
January 29, 2021

Berkeley
UNIVERSITY OF CALIFORNIA

FPGAs: Building Blocks of Logic

FPGA Structure

- Array of ”Logic Cells” and interconnect
- What are “Logic Cells” exactly?
  - How to implement every possible logic function in finite space?
  - How to adapt to any N-bit wide input?



Berkeley
UNIVERSITY OF CALIFORNIA

Truth Tables

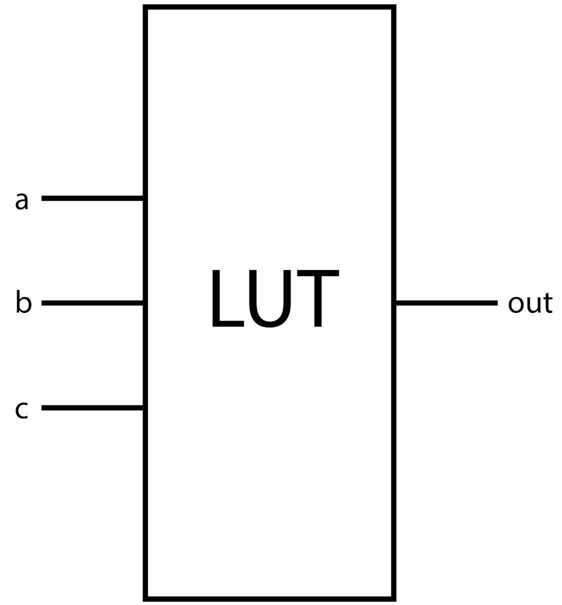
- Completely characterizes logic function
  - Any N-input function requires 2^N rows to fully define
- Map input to output for all possible inputs
  - Could we represent logic functions this way?

c	b	a	out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Look-Up Tables (LUTs)

- Like a hardware truth table
- Map each input to corresponding output

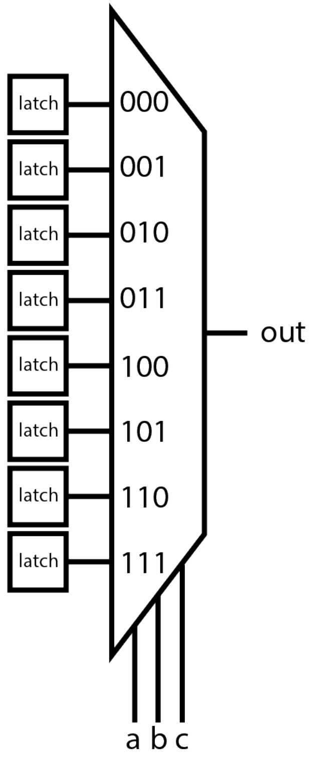
c	b	a	out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



Look-Up Tables (LUTs)

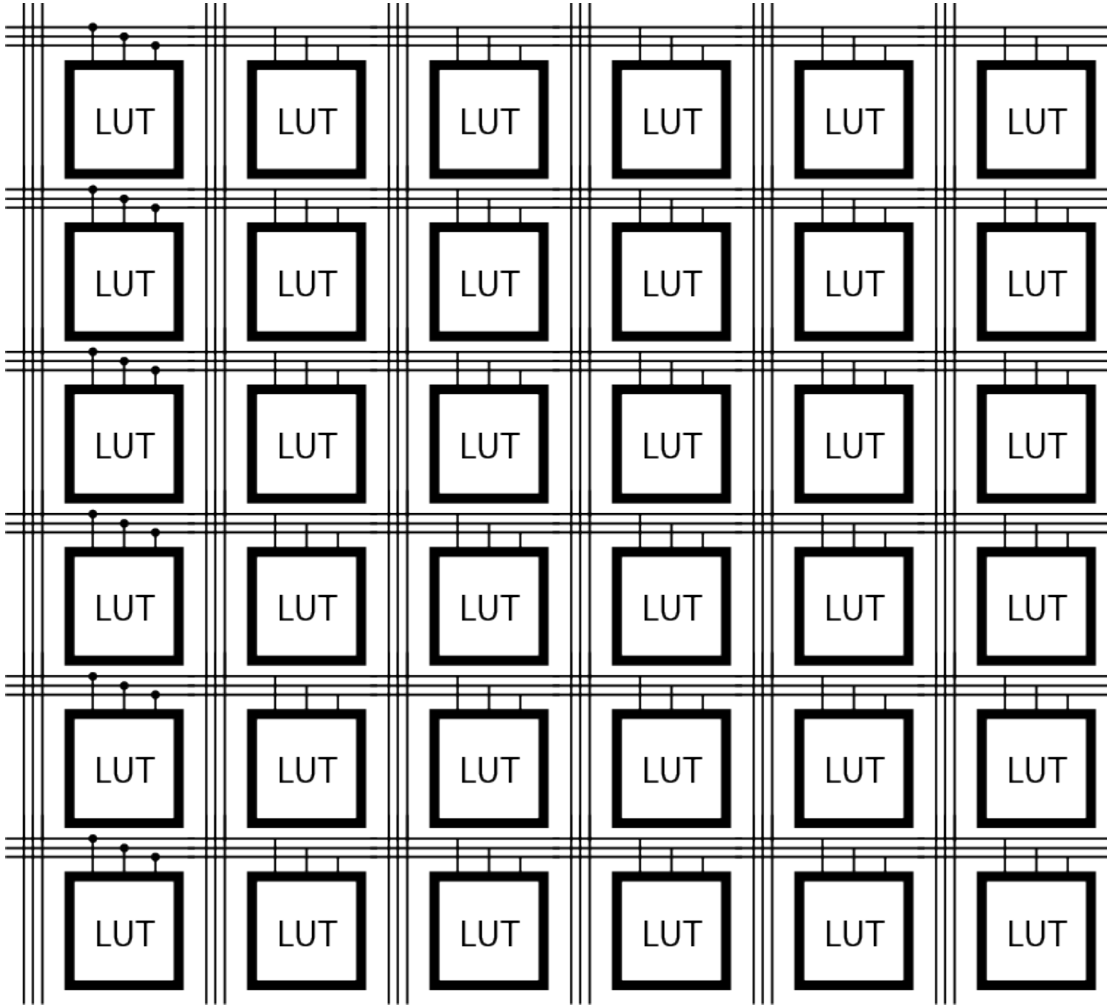
- Like a hardware truth table
- Map each input to corresponding output
- Easy way to implement
  - Use mux with programmable latches on each input
  - Program Latch to correspond to expected output
  - Select output with inputs to LUT

c	b	a	out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



Proto-FPGA

- Array of LUTs and interconnect
- Here’s a proto-FPGA of 3-input LUTs
  - Can perform any combination of 3-input logic functions!
- What if we want to have a 4-input function?



Berkeley
UNIVERSITY OF CALIFORNIA

Building Bigger LUTs

- Consider a 4-input LUT
  - This one is a 4-input XOR
- How to build this out of 3 input LUTs?

d	c	b	a	out
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

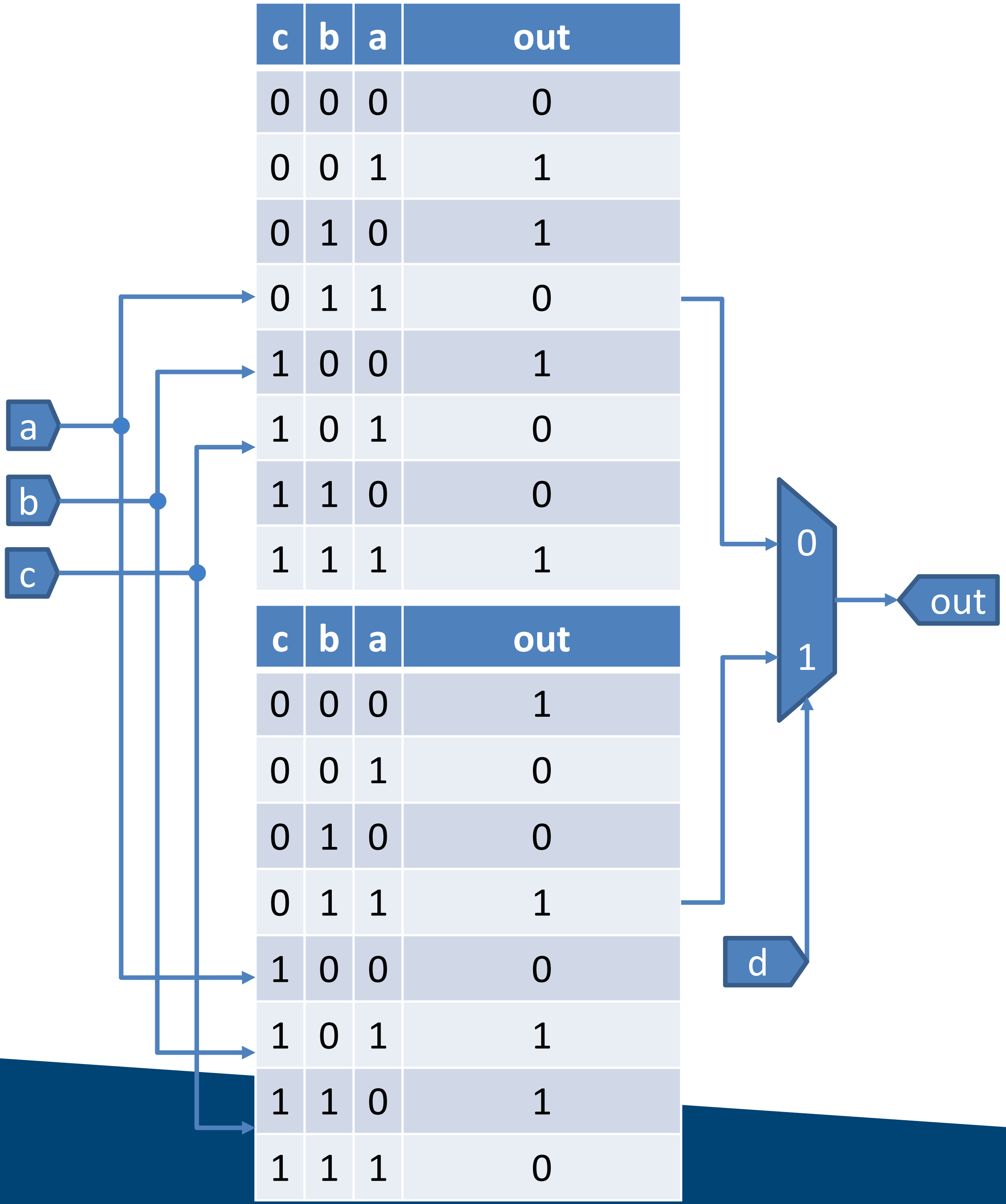
Building Bigger LUTs

- Consider a 4-input LUT
  - This one is a 4-input XOR
- How to build this out of 3 input LUTs?
- Notice how the LUT depends on d
  - Can split into d=0 and d=1 halves
  - abc inputs look identical!

	d	c	b	a	out
d = 0	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	1
	0	0	1	1	0
	0	1	0	0	1
	0	1	0	1	0
	0	1	1	0	0
	0	1	1	1	1
d = 1	1	0	0	0	1
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	1
	1	1	0	0	0
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	0

Building Bigger LUTs

- Consider a 4-input LUT
  - This one is a 4-input XOR
- How to build this out of 3 input LUTs?
- Notice how the LUT depends on d
  - Can split into d=0 and d=1 halves
  - abc inputs look identical!



c	b	a	out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

c	b	a	out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

LUT Caveats

- Can implement any logic function as a LUT
  - **Just because you can doesn't mean you should**
- Ex: 64-inputs require $2^{64}=1.84\text{x}10^{19}$ lines of LUT!
  - Bit width common in arithmetic or encoders
    - Might use LUTs for sub-blocks
  - LUT not most efficient way to implement a function
    - But it is very straightforward

Boolean Algebra

Berkeley
UNIVERSITY OF CALIFORNIA

Functional Completeness

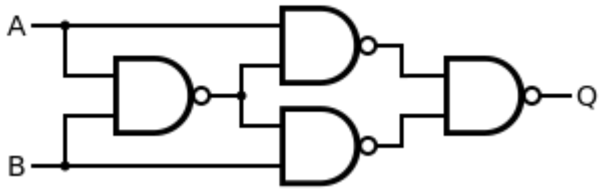
- How would you build an XOR gate out of only ANDs and ORs?

Functional Completeness

- How would you build an XOR gate out of only ANDs and ORs?
  - Spoiler: You can't
- Need a NOT for functional completeness
- Are NANDs functionally complete? Can you make an XOR out of them?

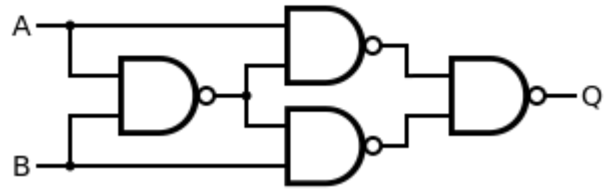
Functional Completeness

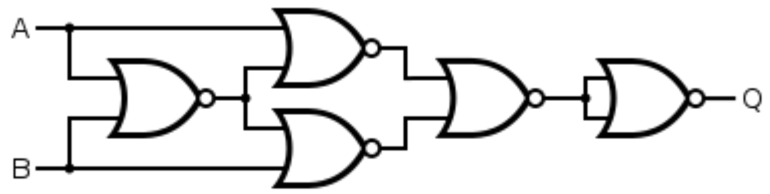
- How would you build an XOR gate out of only ANDs and ORs?
  - Spoiler: You can't
- Need a NOT for functional completeness
- Are NANDs functionally complete? Can you make an XOR out of them?
- What about NORs?



Functional Completeness

- How would you build an XOR gate out of only ANDs and ORs?
  - Spoiler: You can't
- Need a NOT for functional completeness
- Are NANDs functionally complete? Can you make an XOR out of them?
- What about NORs?





Berkeley
UNIVERSITY OF CALIFORNIA

Logic as Math

- Basic operators
  - AND (*, ∧)
  - OR (+, ∨)
  - NOT(¬, ‘, !, ~, or “bar” - ex: $\bar{a}$)
- Order of Operations
  - Similar to arithmetic, AND (*) is done before OR (+), NOT takes precedence over AND
    - $a'b + bc = \big((a') \cdot b\big) + (b \cdot c)$
- Laws and Properties
  - Boolean Algebra has its own set of laws and properties to simply expressions

Berkeley
UNIVERSITY OF CALIFORNIA

Properties

- Properties listed in Lecture 6 Slides
  - Useful for transforming expressions to be easier to simplify
- Here is a selection of some useful ones

$ab = bc$	$a + b = b + a$	$a'' = a$	
$(ab)c = a(bc)$	$(a + b) + c = a + (b + c)$	$a \cdot 0 = 0$	$a + 1 = 1$
$a(b + c) = ab + ac$	$a + bc = (a + b)(a + c)$	$a \cdot 1 = a$	$a + 0 = a$
$(a + b + \cdots + c)' = a'b' \cdots c'$	$(ab \cdots c)' = a' + b' + \cdots + c'$	$a \cdot a = a$	$a + a = a$
$ab' + ab = a(b' + b) = a(1) = a$		$a \cdot \bar{a} = 0$	$a + \bar{a} = 1$

Canonical Forms

- Every Boolean expression can be expressed in one of these forms
- Sum-of-Products (SOP)
  - Sum (OR) of series of products (AND)
    - Ex: $a'b + bc + acd + c'd + b'd$
    - Each product sometimes referred to as a “minterm” if SOP in most simplified form
- Product-of-Sums (POS)
  - Product (AND) of series of sums (OR)
    - Ex: $(a + b)(b + c)(a + c' + d)(b + d)$
    - Each sum sometimes referred to as a “maxterm” if POS in most simplified form
- Can use different methods to simplify down to one of these two forms
  - Karnaugh maps (K-maps) are one such method

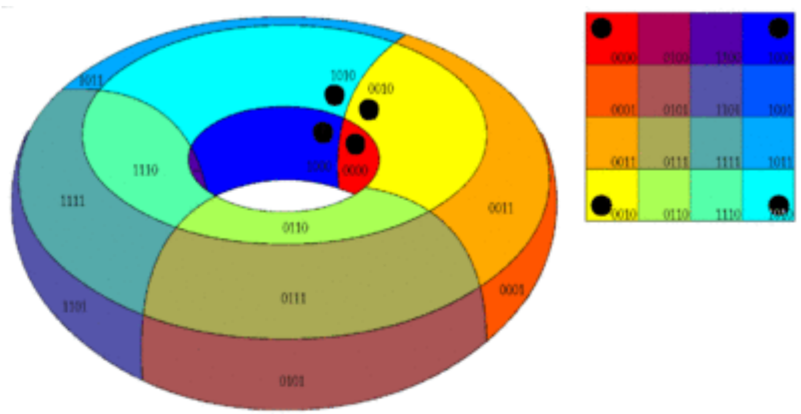
Karnaugh Maps

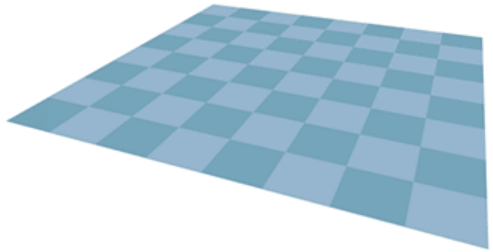
- Visualize Truth Table
  - Keep "adjacent" terms nearby
    - Adjacency means only 1 bit changes between them
  - Wikipedia has a decent illustration of adjacency
    - https://en.wikipedia.org/wiki/Karnaugh_map#Karnaugh_map
- Can use this to find either SOP or POS representation of function

CD \ AB	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Karnaugh Maps

- Visualize Truth Table
  - Keep "adjacent" terms nearby
    - Adjacency means only 1 bit changes between them
  - Wikipedia has a decent illustration of adjacency
    - https://en.wikipedia.org/wiki/Karnaugh_map#Karnaugh_map
- Can use this to find either SOP or POS representation of function





Gray Code

- Each adjacent term in sequence only differs by 1 bit
  - 00, 01, 11, 10
- Mapping truth table by Gray code leads to K-map adjacency
  - Each term differs in input by only 1 bit from its neighbors

AB

CD	A′B′	A′B	AB	A′B
C′D′	0	1	0	1
C′D	1	0	1	0
CD	0	1	0	1
CD′	1	0	1	0

Gray Code

- Each adjacent term in sequence only differs by 1 bit
  - 00, 01, 11, 10
- Mapping truth table by Gray code leads to K-map adjacency
  - Each term differs in input by only 1 bit from its neighbors
- Can also think of it like the K-map is tiled on all sides
  - Edges "wrap around"
    - Like a Pac-man stage

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0