EECS 151/251A SP2022 Discussion #2

GSI: DIMA NIKIFOROV, YIKUAN CHEN

Agenda

- Administrivia
- More Verilog
- Testbenches
- Combinational Logic

Administrivia

- Thoughts on hybrid instruction/labs?
- Homework 2 posted

More Verilog

Blocking vs. Nonblocking

Blocking:

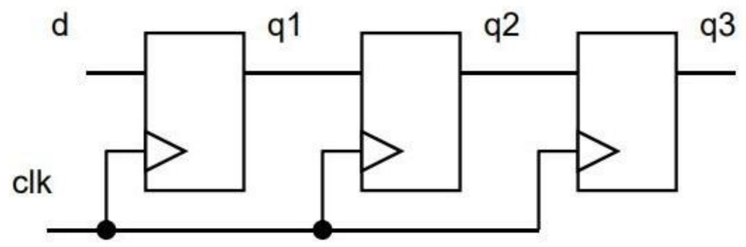
```
reg c, out;
wire a, b, d;
always @(*) begin
    c = a | b;
    out = c & d;
end
```

Non-Blocking:

```
reg c, out;
wire a, b, clk;
always @(posedge clk) begin
    //use val of ‘out’ before clk edge
    c <= out | a;
    //use val of ‘c’ before clk edge
    out <= c & b;
end
```

- Don’t mix blocking and nonblocking!
- When would you use either one?

Race Conditions: Synthesis vs. Simulation



Want: a register pipeline

Determine:
1. Does it *synthesize* correctly?
2. Does it *simulate* correctly?
   - Note: always blocks may simulate in any order
3. Is it good coding practice?

Candidate #1:
```
always @(posedge clk) begin
  q1 = d;
  q2 = q1;
  q3 = q2;
end
```

Candidate #2:
```
always @(posedge clk) begin
  q3 = q2;
  q2 = q1;
  q1 = d;
end
```

Candidate #3:
```
always @(posedge clk) begin
  q1 <= d;
   q2 <= q1;
   q3 <= q2;
end
```

Candidate #4:
```
always @(posedge clk) q1 = d;
always @(posedge clk) q2 = q1;
always @(posedge clk) q3 = q2;
```

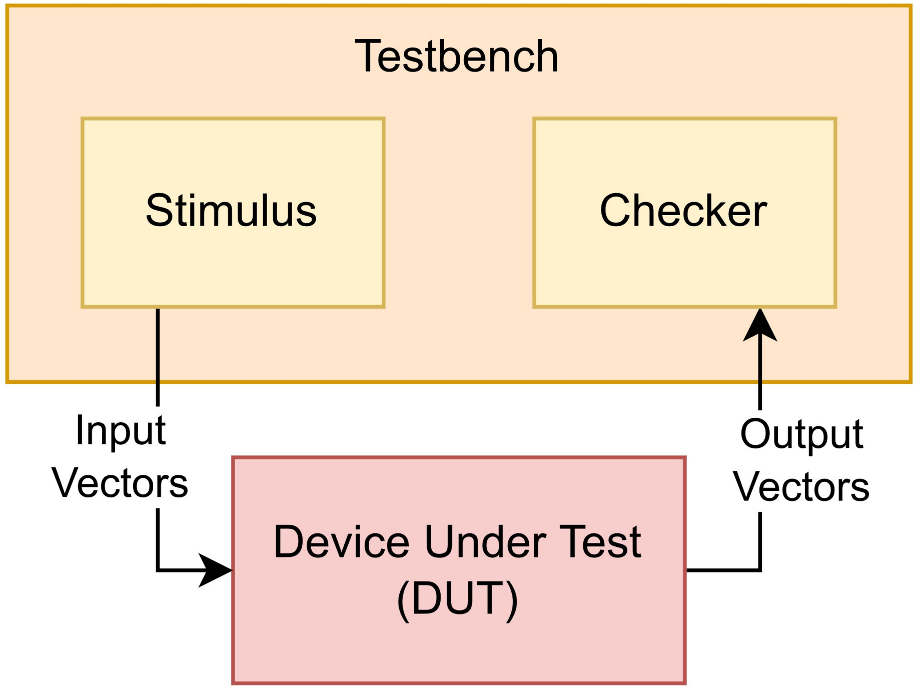
Candidate #5:
```
always @(posedge clk) q3 = q2;
always @(posedge clk) q2 = q1;
always @(posedge clk) q1 = d;
```

Candidate #6:
```
always @(posedge clk) q1 <= d;
always @(posedge clk) q2 <= q1;
always @(posedge clk) q3 <= q2;
```

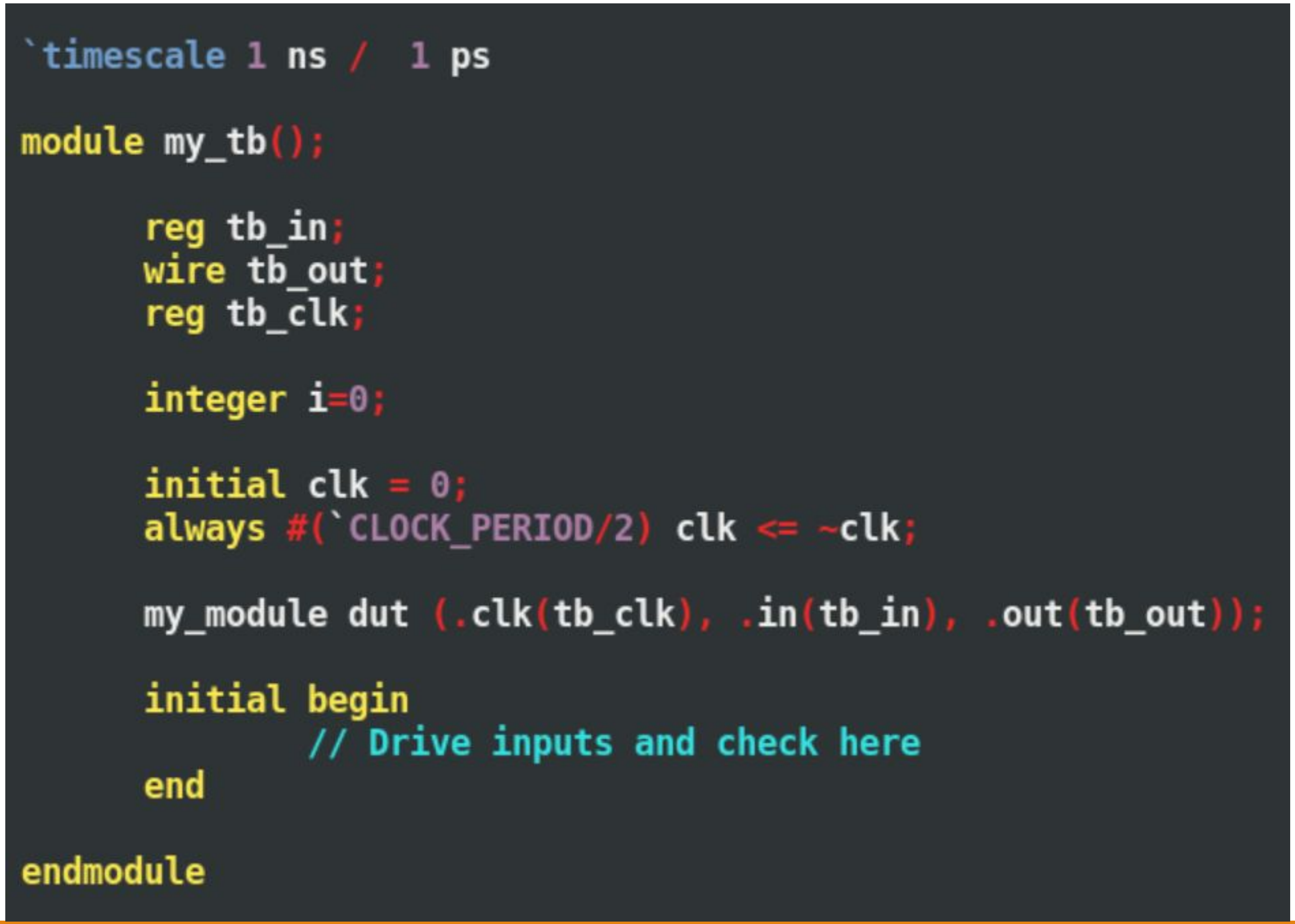
Testbenches

What’s a Testbench?

- Tool to verify that design behaves as specified
- Generate inputs to drive design
- Compare outputs against expected results



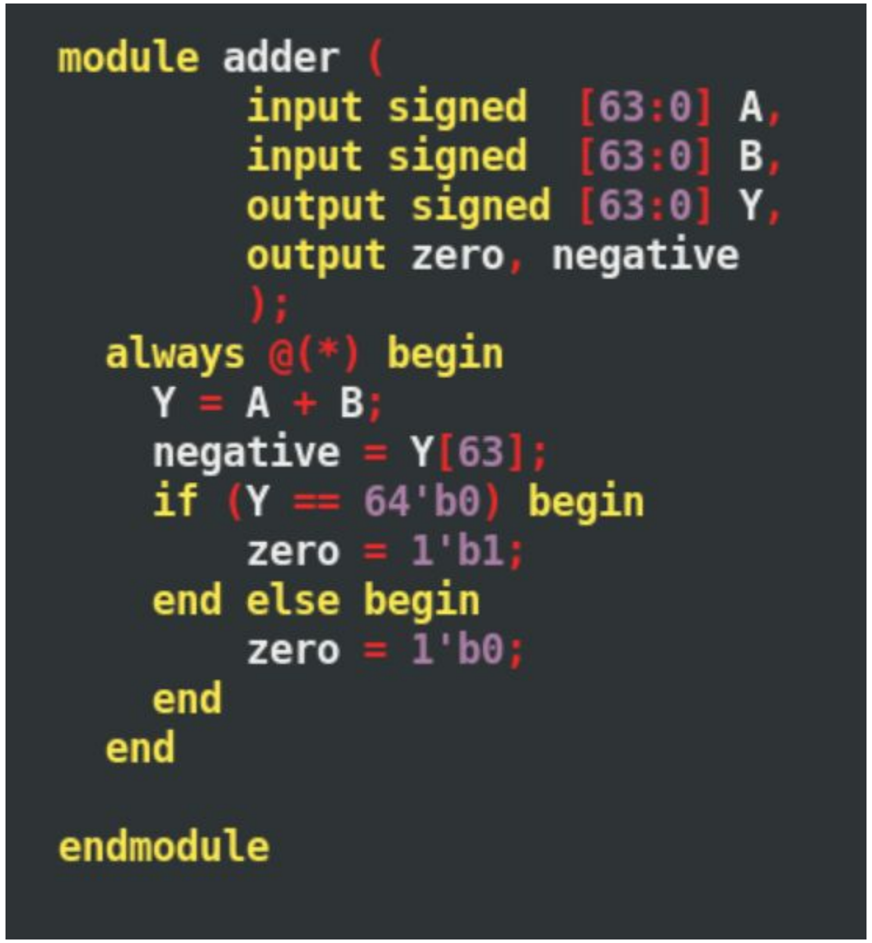
Example Testbench



```
`timescale 1 ns /  1 ps

module my_tb();

      reg tb_in;
      wire tb_out;
      reg tb_clk;

      integer i=0;

      initial clk = 0;
      always #(`CLOCK_PERIOD/2) clk <= ~clk;

      my_module dut (.clk(tb_clk), .in(tb_in), .out(tb_out));

      initial begin
              // Drive inputs and check here
      end

endmodule
```

What Makes a Good Testbench?

- Code coverage:
  - Statements
  - Branches
  - Toggles
  - States
- Functional coverage
  - Features/Requirements



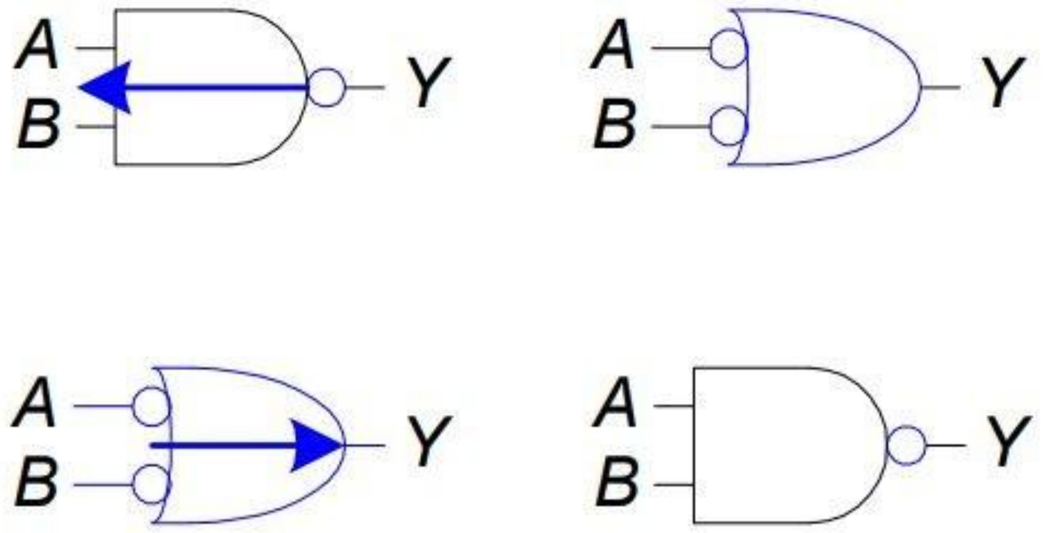
```
module adder (
        input signed  [63:0] A,
        input signed  [63:0] B,
        output signed [63:0] Y,
        output zero, negative
        );
  always @(*) begin
    Y = A + B;
    negative = Y[63];
    if (Y == 64'b0) begin
        zero = 1'b1;
    end else begin
        zero = 1'b0;
    end
  end

endmodule
```

Combinational Logic

Boolean Algebra: DeMorgan’s

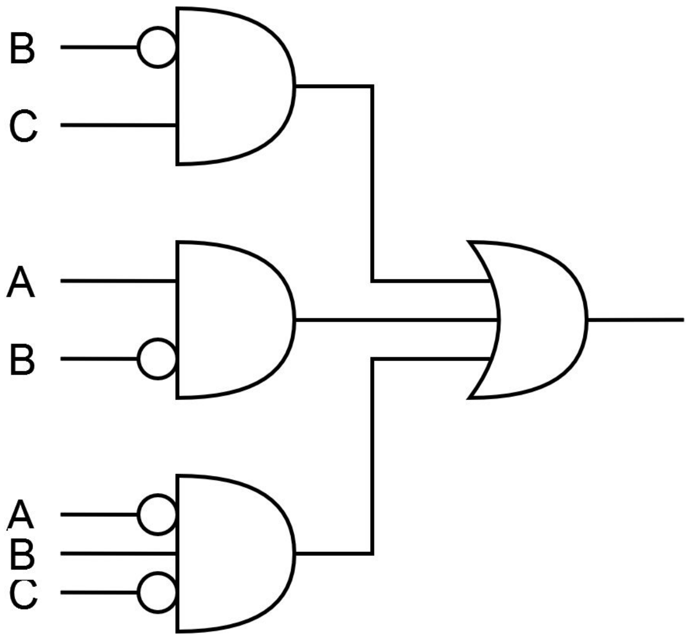
- First step towards logic simplification
- Recall: (x+y)’ = x’y’ , (xy)’ = x’+y’
- Bubble = inversion (NOT)
- Steps for a single gate:

1. Swap AND for OR & vice versa
2. Backward pushing: add bubbles to inp
3. Forward pushing: add bubbles to outp

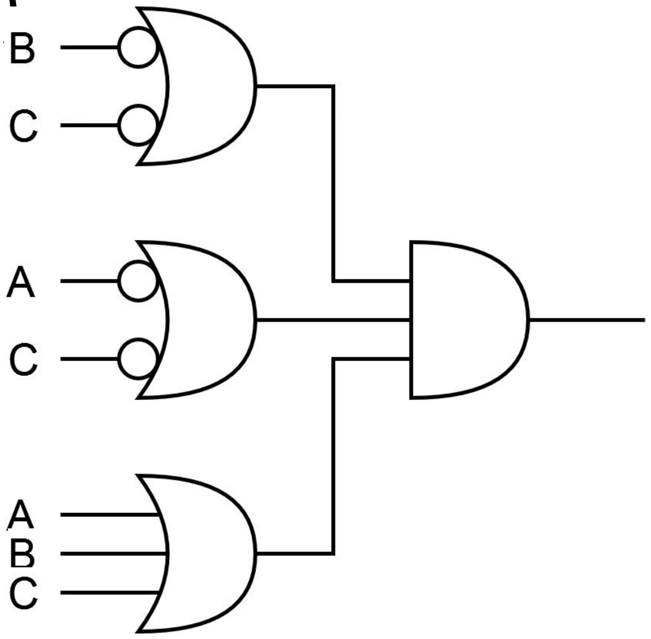


Canonical Forms

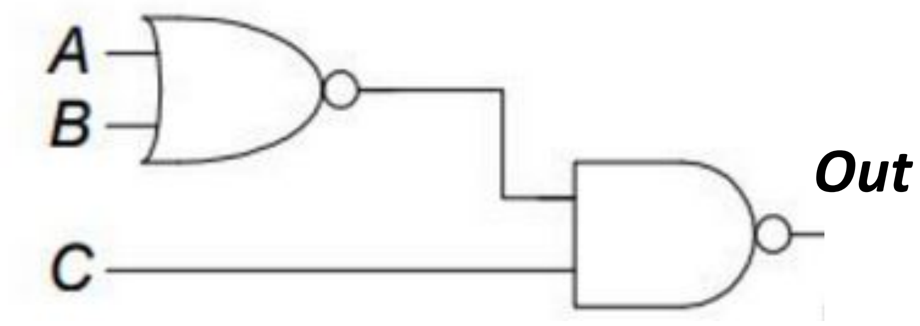
- **Sum of Products (SoP):**
  - $\overline{B}C + A\overline{B} + \overline{A}B\overline{C}$



- **Sum of Products (SoP):**
  - $(\overline{B} + \overline{C})\ (\overline{A} + \overline{B}\ )\ (A + B + C)$



Truth Tables



A	*B*	*C*	*Out*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Truth Tables

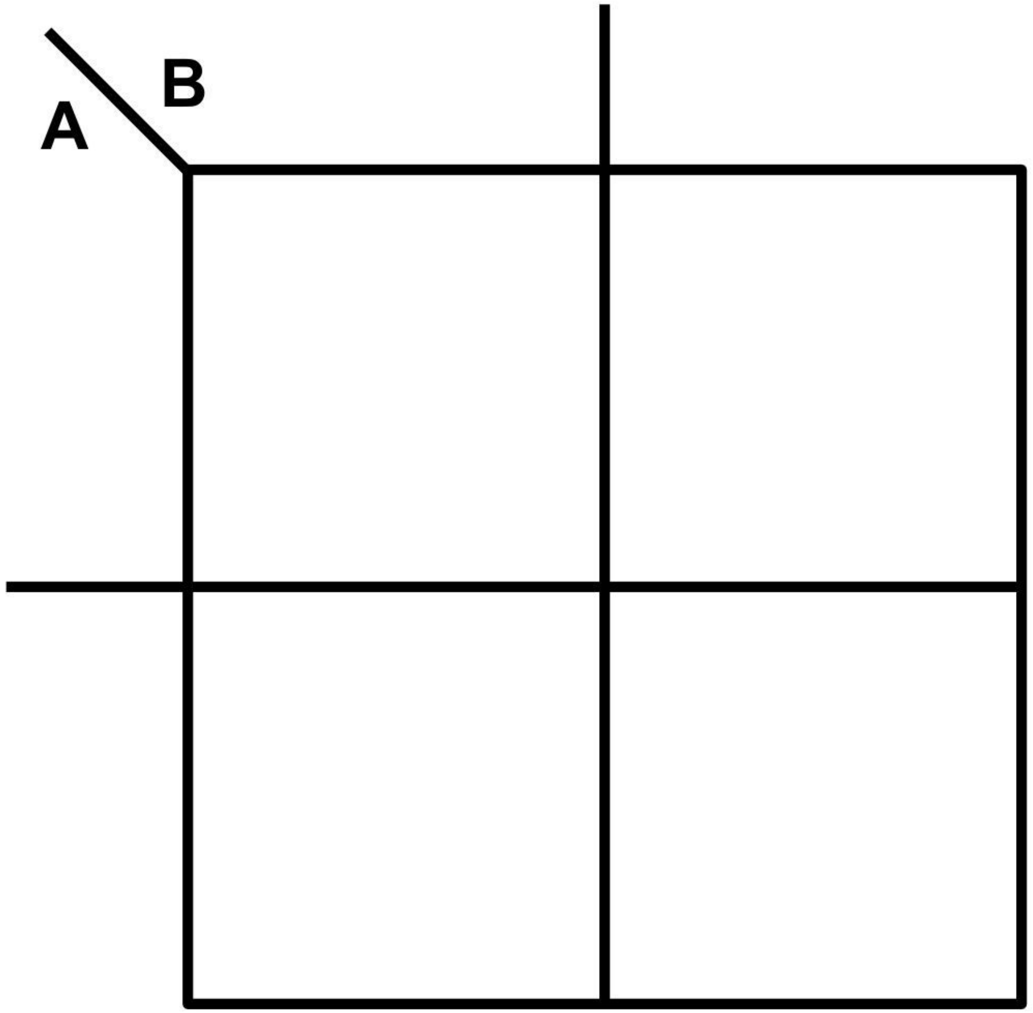
A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

SoP

PoS

2-input K-Map

$F(A,B) = \overline{A}B + AB + A\overline{B}$



4-input K-Map

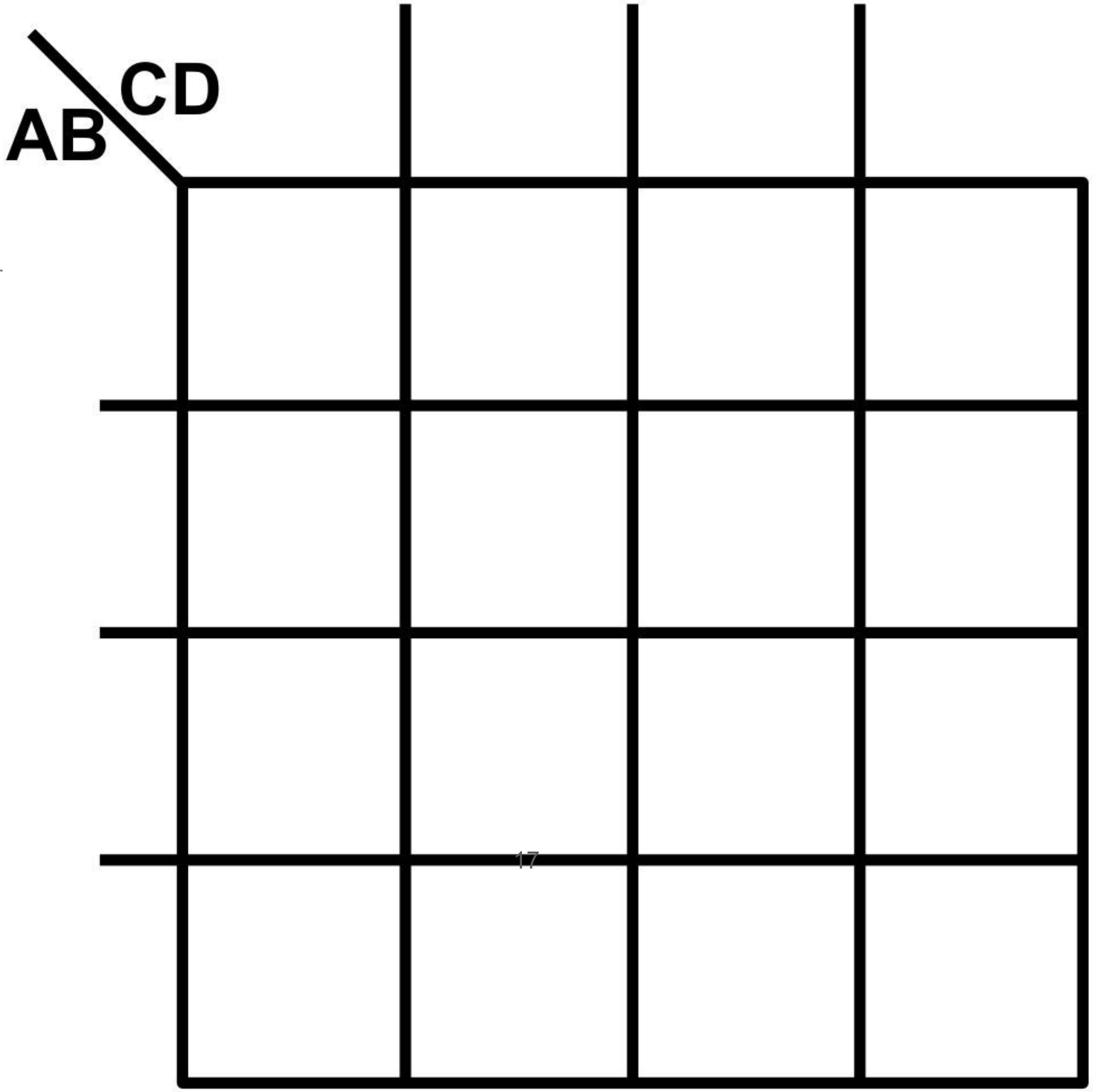
F(A,B) = $\overline{A}\overline{B}\overline{C}\overline{D}$ + $\overline{A}\overline{B}C\overline{D}$ +

$\overline{A}B\overline{C}\overline{D}$ + $\overline{A}B\overline{C}D$ +

$AB\overline{C}\overline{D}$ + $ABC\overline{D}$ +

$A\overline{B}CD$ + $A\overline{B}\overline{C}\overline{D}$ +

$ABCD$



Questions?