

# ECE253 Midterm Cheatsheet

## Boolean Algebra

De Morgan's Theorem tells us

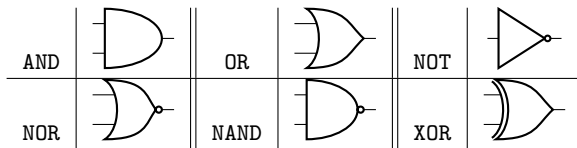
$$\overline{x \cdot y} = \overline{x} + \overline{y}, \quad \overline{x + y} = \overline{x} \cdot \overline{y} \quad (1)$$

Inverting the inputs to an **or** gate is the same as inverting the outputs to an **and** gate, and the other way around.

We also have:

- $(x + y)(y + z)(\overline{x} + z) = (x + y)(\overline{x} + z)$
- $x + yz = (x + y)(x + z)$
- $x + xy = x$  (Absorption)
- $xy + x\overline{y} = x$  (Combining)
- $(x + y)(x + \overline{y}) = x$
- $x + \overline{x}y = x + y$
- $x(\overline{x} + y) = xy$
- $xy + yz + z\overline{x} = xy + z\overline{x}$  (Consensus)

## Gates



## SOPs and POSs

We can create boolean algebra expressions for truth tables.

**Minterm:** Corresponds to each row of truth table, i.e.  $m_3 = \overline{x_2}x_1x_0$  such that when  $3 = 0b011$  is substituted in,  $m_3 = 1$  and  $m_3 = 0$  otherwise.

**Maxterm:** They give  $M_i = 0$  if and only if the input is  $i$ . For example,  $M_3 = x_2 + \overline{x_1} + \overline{x_0}$ .

**SOP and POS:** Truth tables can be represented as a sum of minterms, or product of maxterms.

## Cost

The cost of a logic circuit is given by

$$\text{cost} = \text{gates} + \text{inputs} \quad (2)$$

If an inversion (**NOT**) is performed on the primary inputs, then it is not included. If it is needed inside the circuit, then the **NOT** gate is included in the cost.

## Karnaugh Map

Method of finding a minimum cost expression: We can map out truth table on a grid for easier pattern recognition. Example of a four variable map is shown below:

		$x_2x_1$			
		00	01	11	10
$x_4x_3$	00	1	1	1	0
	01	1	1	1	0
	11	0	0	1	1
	10	0	0	1	1

and the representation is  $\overline{x_2} \cdot \overline{x_4} + x_2 \cdot x_1 + \overline{x_4} \cdot x_2$  when using *minterms*. To use *maxterms*, we take the intersection of sets that don't include blocks of 0s. For example,  $(\overline{x_2} \cdot \overline{x_1})(\overline{x_2} + x_1 + x_4)$ . Some *rules*:

- Side lengths should be powers of 2 and be as large as possible.
- Use **graycoding**: adjacent rows/columns should share one bit.

Some *definitions*:

- **Literal:** variables in a product term:  $x_1\overline{x_2}x_3$  has three literals.
- **Implicant:** a product term that indicates the input valuation(s) for which a given function is equal to 1.
- **Prime Implicant:** an implicant that cannot be combined into another implicant with fewer literals. *They are as big as possible.*
- **Cover:** A collection of implicants that account for all valuations for which function equals 1.
- **Essential Prime Implicant:** A prime implicant that includes a minterm not included in any other prime implicant. *They contain at least one minterm not covered by another prime implicant.*

In the above example,  $\overline{x_2} \cdot \overline{x_4} + x_2x_1 + \overline{x_4}x_2$  are prime implicants.

## Minimization Procedure

1. Generate all prime implicants for given function  $f$
2. Find the set of essential prime implicants
3. If the set of essential prime implicants cover all valuations for which  $f = 1$ , then this set is the desired cover. Otherwise, determine the nonessential prime implicants that should be added.

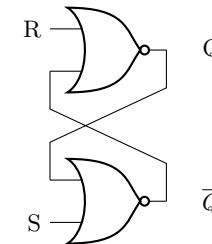
## Common Logic Gates

To save space, boolean expressions will be written instead of drawing diagrams. You should be familiar with how to construct diagrams from expressions.

- **Mux 2→1:**  $\text{mux2to1}(s, x_0, x_1) = \overline{s}x_0 + sx_1$
- **Mux 4→1:**  $\text{mux4to1}(s, x) = \text{mux2to1}(s1, \text{mux2to1}(s0, x_0, x_1), \text{mux2to1}(s0, x_2, x_3))$
- **Not:**  $\text{not}(x) = \text{nand}(x, x) = \text{nor}(x, x)$

## RS Latch

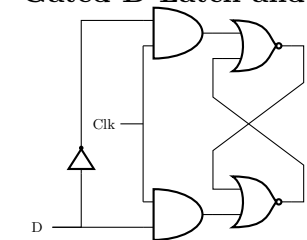
Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled **NOR** gates.



S	R	Q	$\overline{Q}$
0	0	0/1	1/0
0	1	0	1
1	0	1	0
1	1	0	0

When  $S = R = 0$ , it stores the last  $Q$  value. In practice, we should not have  $S = R = 1$ .

## Gated D Latch and Clock Signal



Clk	S	R	$Q(t+1)$
0	x	x	$Q(t)$
1	0	0	$Q(t)$
1	0	1	0
1	1	0	1
1	1	1	1

Where the Clk = 1 cases refer to **retain**, **reset**, **set**, and last one is not used.

## D Flip Flops

Consists of two gated D latches, connected in series and both connected to the same clock. However, clock input for the first D latch is inverted.

- When the clock rises up,  $Q$  stores value of  $D$ .

**Registers:** Multiple flip flops connected together.

## Verilog

### Logic Operators

bitwise AND	&	bitwise OR	
bitwise NAND	~&	bitwise NOR	~!
bitwise XOR	^	bitwise XNOR	^^
logical negation	!	bitwise negation	~
concatenation	{}	replication	{}}

- reduction operators are put at the start and output a scalar.
- bitwise operators
- blocking assignment `=:` executed in the order they are specified.
- Nonblock assignments `<=` executed in parallel.

### Minimal Example

```
module mux(MuxSelect, Input, Out);
    input [7:0] Input;
    input [2:0] MuxSelect;
    output Out;

    reg Out; // declare output for always block

    always @(*) // declare always block
    begin
        case (MuxSelect[2:0]) // start case statement
            3'b000: Out = Input[0]; // case 0
            3'b001: Out = Input[1]; // case 1
            3'b010: Out = Input[2]; // case 2
            3'b011: Out = Input[3]; // case 3
            3'b100: Out = Input[4]; // case 4
            3'b101: Out = Input[5]; // case 5
            3'b110: Out = Input[6]; // case 6
            default: Out = 1'bx; // default case
        endcase
    end
endmodule
```

### Half Adder

```
module HA(x, y, s, c);
    input x, y;
    output s, c;

    assign s = x^y;
    assign c = x&y;
endmodule
```

### Full Adder

```
module FA(a, b, c_in, s_out, c_out);
    input a, b, c_in;
    output s_out, c_out;
    wire w1, w2, w3;

    HA u0(.x(a), .y(b), .s(w1), .c(w2));
    HA u1(.x(c_in), .y(w1), .s(s_out), .c(w3));

    assign c_out = w2|w3;
endmodule
```

### D Latch

```
module D-latch(D, clk, Q);
    input D, clk;
    output reg Q;

    always@(D, clk)
    begin
        if (clk == 1'b1)
            Q = D;
    end
endmodule
```

### Flip Flop

```
module D-ff(D, clk, Q);
    input D, clk;
    output reg Q;
    always@(posedge clk)
    begin
        Q <= D; // use <= operator
    end
endmodule
```

### Flip Flop (stores on both edges)

```
module DDR (input c, input D, output Q) ;
    reg p, n;
    always @ (posedge c)
        p <= D;
    always @ (negedge c)
        n <= D;
    assign Q <= c ? p : n;
endmodule
```

### Registers

```
module reg8(D, clk, Q);
    input clock;
```

```
    input [7:0] D;
    output reg[7:0] Q;
    always@(posedge clock)
        Q <= D;
endmodule
```

### Frequency Dividers

- To half the frequency, connect  $\overline{Q}$  to  $D$  on the same gated D latch.
- To quarter the frequency, connect  $\overline{Q}$  to the clock of the next gated D latch (which is set up the same as the half frequency case).
- To reduce frequency by  $2k$ , connect  $k$  D latches connected in series ( $D$  to  $Q$ ) and to the same clock. First  $D$  is connected to last  $\overline{Q}$ . The last  $Q$  will have a reduced frequency of  $2k$ .

### Other Things

- Use minterms when you have to use NAND gates and maxterms when you have to use NOR gates.
- When converting expressions to its dual, it's often helpful to negate expressions twice, or draw out the logic circuit.
- XOR acts as modular arithmetic.
- On the DE1-SoC board, hex thing is red if 0 and white if 1.

### Add Extra Things Below