ECE253 Midterm Cheatsheet

Boolean Algebra

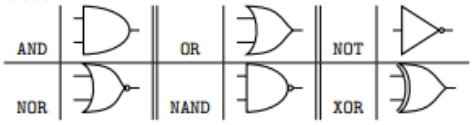
De Morgan's Theorem tells us

$$\overline{x \cdot y} = \overline{x} + \overline{y}, \qquad \overline{x + y} = \overline{x} \cdot \overline{y} \tag{1}$$

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.

- 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.

Cost

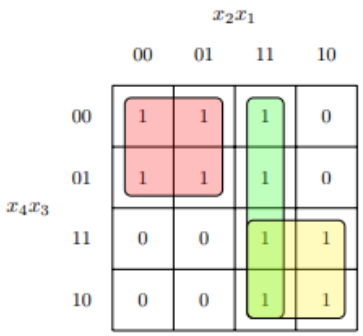
The cost of a logic circuit is given by

$$\text{cost} = \text{gates} + \text{inputs} \tag{2}$$

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:



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. Determine the nonessential prime implicants that should be added.

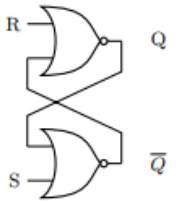
Common Logic Gates

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:** `mux2to1`$(s, x_0, x_1) = \overline{s}x_0 + sx_1$
- **Mux 4→1:** `mux4to1`$(s, x) =$ `mux2to1`$(s1,$ `mux2to1`$(s0, x0, x1),$ `mux2to1`$(s0, x2, x3))$
- **Not:** `not(x)=nand(x,x)=nor(x,x)`
- `XOR` acts as modular arithmetic.
- Multiplexers are functionally complete. `AND` $= mux(x, y, 1)$, `OR` $= mux(x, 0, y)$.

RS Latch

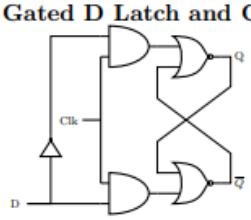
Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled `NOR` gates.



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	D	$Q(t+1)$
0	x	Q(t)
1	0	D

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.

Table 2.1 Axioms of Boolean algebra

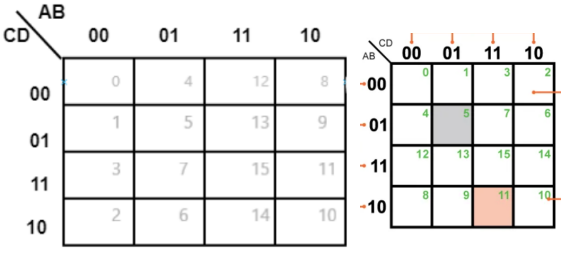
	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= (B \bullet C) + (\overline{B} \bullet D)$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B}_0 + \overline{B}_1 + \overline{B}_2 \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B}_0 \bullet \overline{B}_1 \bullet \overline{B}_2 \ldots)$	De Morgan's Theorem



K-map. Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

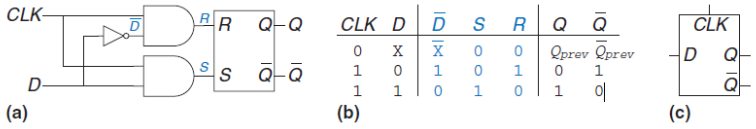
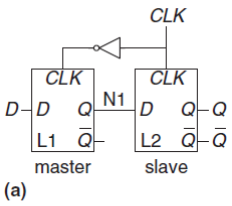


Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

An *enabled flip-flop* adds another input called *EN* or *ENABLE* to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

A *resettable flip-flop* adds another input, called *RESET*. When *RESET* is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When *RESET* is TRUE, the resettable flip-flop ignores *D* and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.



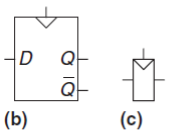
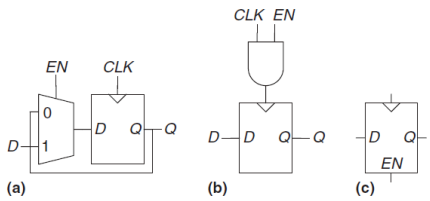
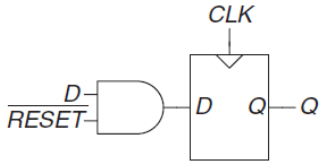
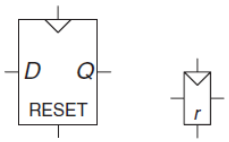


Figure 3.8 D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol





(a)



(b) (c)

Figure 3.11 Synchronously resettable flip-flop: (a) schematic, (b, c) symbols

Figure 3.10 Enabled flip-flop: (a, b) schematics, (c) symbol

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as *RESET* becomes TRUE, independent of *CLK*.

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 [4:0] 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
      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) Q <= D; // use <= operator
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
```

ModelSim Do Files

```
# set working dir, where compiled verilog goes
vlib work
# compile all verilog modules in mux.v to working
# dir could also have multiple verilog files
vlog mux.v
#load simulation using mux as the
# top level simulation module
vsim mux
#log all signals and add some signals to
# waveform window
log {/*}
# add wave {/*} would add all items in
# top level simulation module
add wave {/*}
# first test case
#set input values using the force command
# signal names need to be in {} brackets
force {SW[0]} 0
force {SW[1]} 0
force {SW[9]} 0
run 10ns
```

ModelSim and Other Lab Things

- FGPA: Field Programmable Gate Array
- To repeat signals, use this syntax:

  ```
  force {MuxSelect[2]} 0 0ns, 1 {4ns} -r 8ns
  ```

  which starts at 0 at 0ns, 1 at 4ns, and repeats every 8 ns.
- On the `DE1-SoC` board, hex thing is red if 0 and white if 1.

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 $\bar{Q}$. The last Q will have a reduced frequency of $2k$.

Add Extra Things Below

```
gates acting on 4 bit busses */
assign y1 = a & b;      // AND
assign y2 = a | b;      // OR
assign y3 = a ^ b;      // XOR
assign y4 = ~(a & b);   // NAND
assign y5 = ~(a | b);   // NOR
```

```
module latch(input  logic      clk,
             input  logic [3:0] d,
             output logic [3:0] q);

 always_latch
  if (clk) q <= d;
endmodule
```