Powers of Two: $2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, $2^4=16$, $2^5=32$, $2^6=64$, $2^7=128$, $2^8=256$, $2^9=512$, $2^{10}=1024$, $2^{11}=2048$, $2^{12}=4096$, $2^{13}=8192$, $2^{14}=16394$, $2^{15}=32768$, $2^{16}=65536$

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Number Conversion: **D→B: 1.** Determine the largest power of 2 that fits into the number. **2.** Then represent the difference with the largest power of 2 that fits into the difference of the two numbers. Repeat steps 1-2. **3.** For the powers of 2 that weren't needed, treat them as 0s, and the rest as 1s.
B→D: 1. Expand the binary code as the full sum. **2.** Add up all the terms.
B→H: 1. Group the bits in 4s from R-to-L. **2.** Add zeros at the left if doesn't make group of 4.

Gates:

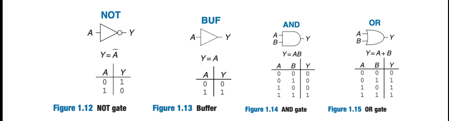


Figure 1.12 NOT gate Figure 1.13 Buffer Figure 1.14 AND gate Figure 1.15 OR gate

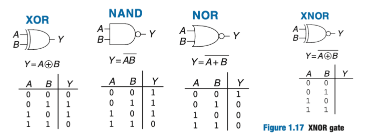


Figure 1.17 XNOR gate

NAND and **NOR** are functionally complete.

SOP: an OR of ANDS. Any SOP circuit can be implemented as NAND.
Minterm: Product term that **outputs to 1** for exactly 1 row of TT. **x_i if x_i=1 & $\bar{x}_i$ if x_i=0.**
POS: AND of ORs. Any POS circuit can be implemented as NOR.
Maxterms: Sum term that **outputs to 0** for exactly 1 row of TT. **x_i if x_i=0 & $\bar{x}_i$ if x_i=1.**

Simplifying Equations:
-Expanding an implicant ($AB=ABC+AB\bar{C}$) useful to be shared with another minterm.
-Use distributive to expand out POS into SOP.
-Duplicate terms using idempotency.
-Check: ****Perfect Induction**** (LS=RS)

-Anytime we need to implement NOR or NAND, use DeMorgans.
Implicant: Any product term included by f.
PI: An implicant for which it is not possible to remove any literal and still have a valid implicant. The **largest group of 1s which can be circled** to cover the particular 1 in K-map.
Cover: Any set of implicants that includes all minterms of a function.
EPI: A PI that covers at least one minterm that is not covered by any other PI.
Procedure for Finding a Min. Cost Cover:
1. Find PIs and identify EPIs (include in cover)
2. Choose other PIs as needed until all minterms are covered, such that the number of literals is minimized. There can be ****multiple min. cost covers**.**

$$Cost = \#\,gates + \#\,inputs$$

K-Map:
1. Use the fewest circles necessary to cover all the 1's.
2. All the squares in each circle must contain 1's.
3. Each circle must span a rectangular block that is a power of 2 squares in each direction.
4. Each circle should be as large as possible.
5. A circle may wrap around the edges of the K-map.
6. A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.
Don't Cares (d): Each don't care minterm can arbitrarily be 0 or 1.

Multiplexers/Decoders:
Mux chooses an output from among several possible inputs, based on the value of a **select** signal. If s==0, then f is controlled by x, else y.
2:1 mux - selects 1 of 2 possible inputs
4:1 mux - select 1 of 4 possible inputs (s would be two bits)

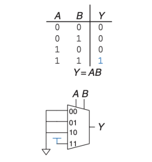
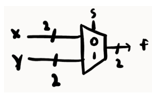
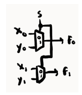


Figure 2.59 4:1 multiplexer implementation of two-input AND function

Mux with 2-bit inputs x,y



s	f_1	f_0
0	x_1	x_0
1	y_1	y_0



Verilog Syntax:

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
Lowest	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
	?:	Conditional

Mux 2-to-1:
module mux2to1(input logic x, y, s, output logic f);
assign f = (~s & x) | (s & y);
endmodule

```
module  mux2to1_2bits(input logic [1:0] x,y, input logic s, output logic [1:0] f);
        assign f[1] = (~s & x[1]) | (s & y[1]);
        assign f[0] = (~s & x[0]) | (s & y[0]);
endmodule
```

Half Adder
```
module HA(input logic a, b, output logic [1:0] s);
        assign s[1] = a & b;
        assign s[0] = a ^ b;
endmodule
```
Full Adder
```
module FA(input logic a, b, cin, output logic s, cout);
        assign s = a ^ b ^ c;
        assign cout = (a & b) | (cin & a) | (cin & b);
endmodule
```
Hierarchical Verilog Code
```
module adder3(input logic [2:0] A, B, input logic cin,
        output logic [2:0] S, output logic cout);
        logic C1, C2; //internal signals
        FA = u0(A[0], B[0], cin, S[0], C1);
        FA = u1(A[1], B[1], C1, S[1], C2);
        FA = u2(A[2], B[2], C2, S[2], cout);
endmodule
```
Segment 7
```
module seg7(input logic [0:1] x, output logic [6:0] H);
        assign H[0] = ~x[1] & x[0];
        assign H[1] = 1'b0; // sets 1 bit to be 0.
        assign H[2] = x[1] & ~x[0]
        assign H[3] = ~x[1] & x[0]
        assign H[4] = x[0]
        assign H[5] = x[1] | x[0]
        assign H[6] = ~x[1]
```

Latches:
RS Latch (Left) & Characteristic Table (Right):

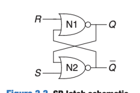
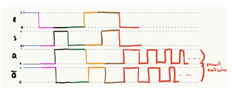


Figure 3.3 SR latch schematic

S	R	Q	$\bar{Q}$	
0	0	$^0/_1$	$^1/_0$	(stored value)
0	1	0	1	
1	0	1	0	
1	1	0	0	(not used in practice)

Inputs S (set) & R (reset), Outputs Q & $\bar{Q}$



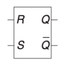
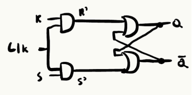
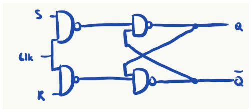


Figure 3.6 SR latch symbol

Gated RS Latch (Clock Input to Dis./Ena. Latch):

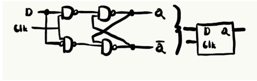
Clk	S	R	Q(t+1)
0	X	X	Q(t)
1	0	0	Q(t)
1	0	1	0
1	1	0	1
1	1	1	Not used





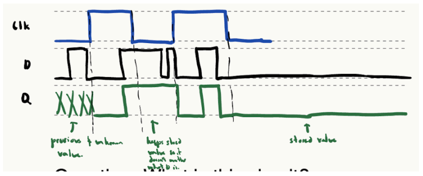
^ Cheaper implementation.
-If **Clk=0, retain previous value**. If **Clk=1, use R and S.**
-When Clk=0, both R' & S' are 0 & the latch is storing the value. Only when Clk is 1, can value be set or reset.
-Problem w/ RS Latch, it can oscillate when R,S=1 to R,S=0.

Gated D Latch:
Let $S = D, R = \bar{D}$, *where D is the data input* ⇒ Impossible to have a situation where R=S=1.

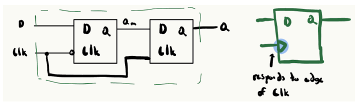


Clk	D	Q(t+1)
0	X	Q(t) (store)
1	0	0 reset (∵ $R=\bar{D}=1$)
1	1	1 set (∵ S=D=1)

-When **clk=1, Q=D**
-When **clk->0, Q will store the present value of D.**

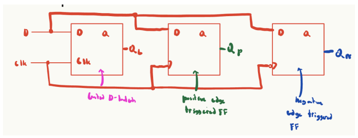


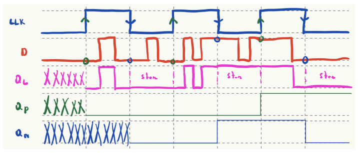
Flip-Flops (Positive Edge Triggered FF):



-C1: **Clk=0: Q_m=0, Q=0/1** (stored value)
-C2: **Clk ("posedge"): Q=Q_m=D** (no longer tracks input changes).
So when Clk changes from 0 to 1, the value of D **at that moment is stored in FF.**

Difference Gated D-Latch, Posedge FF, Negedge FF





Register: Synonym for FF.
Verilog for D-Latch:

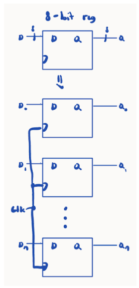
```
module D_latch(input logic D, clk, output logic Q);
        always_latch
              if (clk==1)
                    Q=D;
endmodule // the latch stores Q when clk is not 1.
```

Note: If code does not specify the value of a signal in some conditions, the compiler will insert a latch to store the value of that signal.
Verilog for FFs:

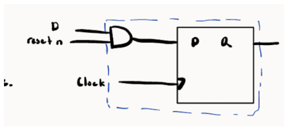
```
module D_FF(input logic D, Clk, output logic Q);
        always_ff@(posedge clk) // (?) is sensitivity list.
                    Q<=D; // <= is an assignment op.
endmodule // use negedge for neg. edge trigg. FF
```

-Any signal assigned a value inside an always block using posedge becomes Q output of a FF.
-When code describes FFs, the assignments should use <=
Verilog for Register:

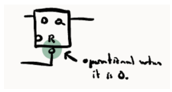
```
module reg8(input logic [7:0] D, input logic clk, output logic [7:0] Q);
        always_ff@(posedge clk)
                    Q<=D;
endmodule
```



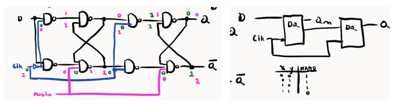
Resets and Enables: Need to reset FF to known value
Resetn: Active low reset (corresponds w/ 0)
Synchronous (with clock edge):



```
module D_FF(input logic D, Clk, resetn, output logic Q);
        always_ff@(posedge Clk) // at clk edge
                    if (resetn==0) // active low reset
                        Q<=1'b0; // Q set to 1 bit as 0.
                    else
                        Q<=D;
endmodule
```



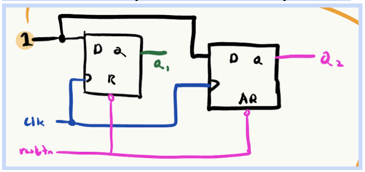
Asynchronous (Independent of clock edge):

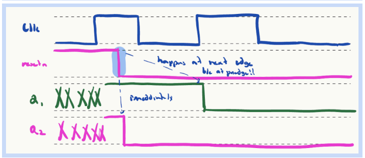


-C1: If **Clk=0, resetn=0, gives** $Q = 0, \overline{Q} = 1$
-C2: If **Clk=1, resetn=0, gives** $Q = 0, \overline{Q} = 1$
-Proves that this is independent of the clock edge.

```
module D_FF(input logic D, clk, resetn, output logic Q);
        always _ff@(posedge clk, negedge resetn)
                    if (resetn==0)
                              Q<=1'b0;
                    else
                              Q<=D;
endmodule
```

-resetn in sensitivity list b/c it can directly affect Q

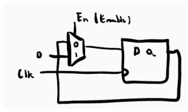


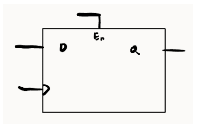


Multi-bit Register With Synchronous Reset

```
module reg8(input logic [7:0] D, input logic resetn, clk, output logic [7:0] Q);
        always_ff@(posedge clk)
                    if(resetn==0)
                              Q<=8'b0;
                    else
                              Q<=D;
```

Enable Input: Determine whether data is load on clock edge.





-Passes new value D to FF if E_n=1, otherwise recycles old state.
FSM:
1. State diagram
2. State-table
3. State assignment (ie. Choose # FFs, and state codes)
4. State-assigned table
5. Synthesize the circuit
-# of FFs & how we define the states is what we changed.

Verilog Code for FSMs:
1. FFs, 2. State-table, 3. Output

```
module FSM(input logic w, clock resetn, output logic z);
        typedef enum logic [1:0] (A,B,C,D) statetype;
        // typedef defines statetype to be 2bit logic values
        // ps,ns are 2-bit and can take A-D.
        // enumerated encodings default to numerical
        // A=00, B=01,C=10,D=11
        statetype ps,ns;
        // FFs
        always_ff @(posedge clock, negedge resetn)
                    if (!resetn) ps <= A;
                    else ps <= ns;
        // State Table
        always_comb
                    case(ps) // change cases on ps
                    A: if (w) ns = B;
                    else ns = A;
                    B: if (w) ns = B;
                    else ns = A;
                    C: if (w) ns = D;
                    else ns = A;
                    D: if (w) ns = B;
                    else ns = C;
                    endcase
        // Output
        assign z = (ps==D); // z is 1 when ps is D.
endmodule
```