CPEN 312

Ruiheng Su 2021

Number representation

We can expand decimal numbers as a weighted sum of powers of ten. For example:

$$1234 = 1(10^3) + 2(10^2) + 3(10^1) + 4(10^0)$$

- A *bit* is a single binary digit (1 or 0)
- A *nibble* is 4 bits, one hex digit
- A *byte* is 8 bits
- A *word* is 16 bits

Hexadecimal:

$$10, 11, 12, 13, 14, 15 \rightarrow \text{A, B, C, D, E, F}$$

Base n → decimal (adding powers of n): for a base $n > 0$ number with k digits $\beta \dots \alpha$:

$$(\beta \dots \alpha)_n = \alpha(n^0) + \cdots + \beta(n^k)$$

Base 2 → decimal (double-dabble):

1. Starting from the MSB, r_i, set $R = 0$.
2. Compute $2R + r_i$
3. Update $R = 2R + r_i$
4. Move on to the next bit, compute $2R + r_{i-1}$
5. Repeat all bits are exhausted; the conversion is equal to the remainder

Decimal → base n (method of successive division): We build the base n number starting from its least significant digit (from the right):

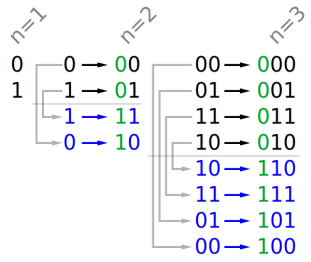
1. Given a number A, find the quotient, q_0, and remainder, r_0, by dividing A by n
2. Divide q_0 by n to compute q_1 and r_1.
3. Divide q_i by n to compute q_{i+1} and r_{i+1}, until $q_{i+1} = 0$
4. The converted number has digits: $(r_i \dots r_0)_n$

Base 8/16 ↔ base 2: Given:

- Base 16: write each digit using 4 bit binary
- Base 8: write each digit using 3 bit binary
- Base 2 to 16: start from the LSB, replace every consecutive 4 digits using hex
- Base 2 to 8: start from the LSB, replace every consecutive 3 digits using octal numbers

Gray code

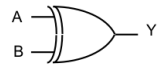
Also called **reflected binary code**. Is an ordering for binary numbers, such that two successive values differ in only a single bit. We can construct them using the reflect-and-prefix method.



Binary Logic

XOR:

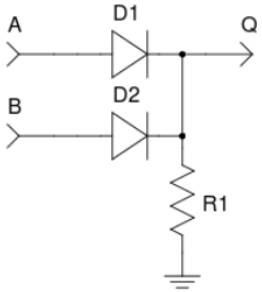
$$\begin{aligned} Y &= A \oplus B \\ &= AB' + A'B \end{aligned}$$



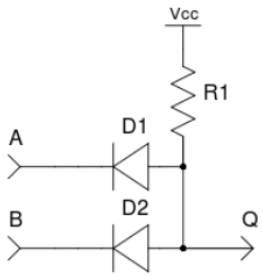
XNOR:

$$\begin{aligned} Y &= (A \oplus B)' \\ &= AB + A'B' \end{aligned}$$

Diode Logic



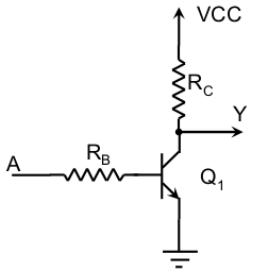
OR Gate



AND Gate

BJT Logic

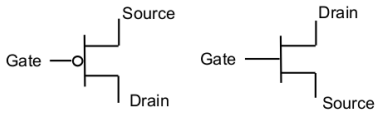
NOT: When A is 1, Q_1 conducts, shorting Y to ground.



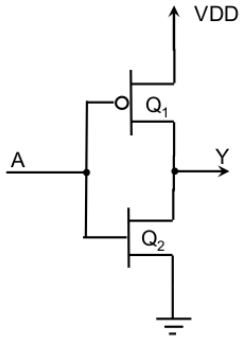
CMOS Logic

P-MOS: Source and drain conducts with logic 0.

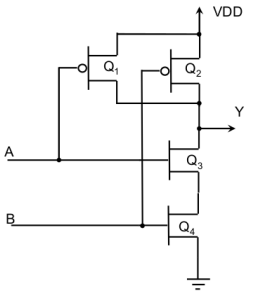
N-MOS: Source and drain conducts with logic 1.



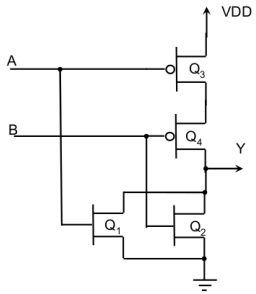
NOT: When $A = 1$, only Q_2 conducts, making $Y = 0$. When $A = 0$, onyl Q_1 conducts, making $Y = 1$



NAND: $Y = 0$ whenever $A = 1$, $B = 1$. Otherwise, one or both of Q_1, Q_2 conducts, and $Y = 1$.



NOR: $Y = 0$ whenever one, or both of A, B is logic 1. $Y = 0$ only even both $A = B = 0$.



Boolean Algebra

Boolean algebra is commutative, associative, and distributive.

- Identity: $A + 0 = A$, $A \cdot 1 = A$
- Redundancy: $A + AB = A$, $A(A + B) = A$

Derived relations:

- $A + A' = 1$, $AA' = 0$
- $A + A = A$, $AA = A$
- $A + 1 = 1$, $A \cdot 0 = 0$

De Morgan's Theorem:

$$(A + B)' = A'B' \qquad (AB)' = A' + B'$$

Canonical forms

Given a true table, of n inputs and k outputs, we want to obtain k boolean functions that determine the k^{th} output as a function of the n inputs.

Sum of minterms: Consider the following truth table

Inputs			Outputs	
c	b	a	X	Y
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Using minterms, we write

$$X = c'b'a + cb'a' + cba$$
$$Y = c'ba + cb'a + cba' + cba$$

Product of maxterms: Consider the following truth table

Inputs			Outputs	
c	b	a	X	Y
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Using maxterms, we write

$$X = (c + b + a)(c + b' + a)(c + b' + a')$$
$$(c' + b + a')(c' + b' + a)$$
$$Y = (c + b + a)(c + b + a')(c + b' + a)$$
$$(c' + b + a)$$

We can still use minterms even if an output, Y, is 1 for more cases than it is 0.

1. Compliement Y for every row in the truth table
2. Write the minterms for Y'
3. Compute $(Y')'$ to recover the maxterms of Y using De Morgan's theorem

Reduction Techniques

Karnaugh maps

Geometrically simplifies a truth table. If there are only a few 1 entries:

1. Negate every entry in the table
2. Group 1s to get the min-terms for Y'
3. Apply De Morgan's theorem to get the max-terms for Y

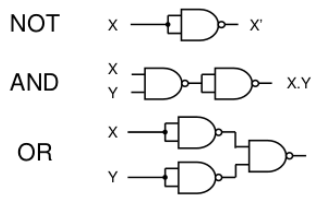
Universal Gates

NAND and NOR can be used to implement any other gate. They are **universal gates**.

NOT ← NAND: $A' = (AA)'$.

AND ← NAND: $AB = \left((AB)'\right)' = \left((AB)'\,(AB)'\right)'$

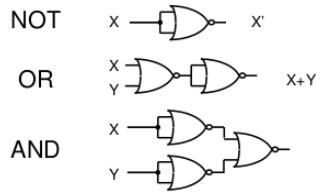
OR ← NAND: $A + B = (A'B')'$



NOT ← NOR: $A' = (A + A)'$.

OR ← NOR: $A + B = (A + B)'' = \left((A + B)'\right)'$

AND ← NOR: $AB = (A + A)''(B + B)''$

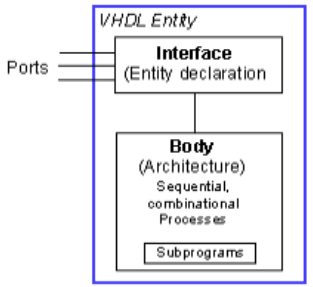


VHDL

Why VHDL:

- High level abstraction
- Easy to debug
- More modular, and easier to understand

Is a **hardware description language** that allows us to programmatically create logic circuits. A script generally consists of



Assignment operators

- <= assignment of signals
- := assignment of variables, and assignment of signals at declaration

Entity

```
entity NAME_OF_ENTITY is [ generic generic_declarations);
    port (
        signal_names: mode type;
        signal_names: mode type;
            :
        signal_names: mode type -- no ";"
    );
end [NAME_OF_ENTITY] ;
```

.....

Architecture

```
architecture architecture_name of NAME_OF_ENTITY is
-- Declarations
        -- components declarations
        -- signal declarations
        -- constant declarations
        -- function declarations
        -- procedure declarations
        -- type declarations
        :
begin
-- Statements
    -- processes
    :
end architecture_name;
```

Signals and variables

Signals are updated at the end of a process.

Variables are updated immediately.

Process

Models sequential statements.

- The sensitivity list is a set of signals to which the process is sensitive
- Change in the value of the signals will cause the process to execute

```
[process_label:] process [ (sensitivity_list) ] [is]
    [ process_declarations]
begin
    -- list of sequential statements such as:
        -- signal assignments
        -- variable assignments
        -- case statement
        -- exit statement
        -- if statement
        -- loop statement
        -- next statement
        -- null statement
        -- procedure call
        -- wait statement
end process [process_label];
```

If statements

```
if condition then
    sequential statements
    [elsif condition then
        sequential statements ]
    [else
        sequential statements ]
end if;
```

Case statements

- Can only be defined within a `process`
- Each choice can be covered only once
- If the “when others" choice is not present, all possible values of the expression must be covered by the set of choices

```
process ( .. ) is
    case expression is
        when choices =>
            sequential statements
        when choices =>
            sequential statements
            -- branches are allowed
        [ when others => sequential statements ]
    end case;
end process;
```

When-else, with-select statements

- Can only be define outside of a `process`

```
architecture a_name of p_name is
    signal a : INTEGER;
    signal b : INTEGER;
    signal c : INTEGER;
    signal d : INTEGER;
begin
    with a select
        b <= 1 when 0,
             2 when 3,
             3 when others;

    c <= 1 when d = 1 else
         2 when d = 2 else
         3;
end a_name;
```

Detecting clock edges

We can use built in functions:

```
if rising_edge(CLOCK) then
    :
end if;
if falling_edge(CLOCK) then
    :
end if;
```

Or use **signal attributes**:

```
if (CLOCK'event and CLOCK='1') then
    :
end if;
```

- `signal_name'event` returns TRUE if an event on the signal called `signal_name` occured
- False otherwise

Alternatively, we can use `WAIT UNTIL` statement:

```
process(CLOCK)
begin
    WAIT UNTIL rising_edge(CLOCK);
    :
end process;
```

Counters

An n-bit counter wraps around. Let Q be a `STD_LOGIC_VECTOR`. We can increment Q by 1 bit:

```
Q <= Q + 1;
```

Q wraps around to zero once all bits are logic 1.

We can do the same using `INTEGER` signals.

Miscellaneous

We can use “`OTHERS`” to set all bits of a `STD_LOGIC_VECTOR` to come value: :

```
Q <= (OTHERS => '0')
```

Binary adder:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity binary_adder is
    port(
        a, b     : in   std_logic_vector(3 downto 0);
        carry_in : in   std_logic;
        sum      : out  std_logic_vector(3 downto 0);
        carry    : out  std_logic
    );
end binary_adder;

architecture a of binary_adder is
    signal sum_temp : std_logic_vector(4 downto 0);
begin

    process(a, b, sum_temp, carry_in)
    begin
        -- "&" concatenates bits
        sum_temp <= ('0' & a)
                    + ('0' & b)
                    + ("0000" & carry_in);
        carry <= sum_temp(4);
        sum <= sum_temp(3 downto 0);
    end process;

end a;
```

Arithmetic

Parity generator

We can construct a parity generator/checker using XOR gates:

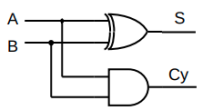
$$\begin{aligned} Y &= A_0 \oplus A_1 \oplus A_2 \oplus A_3 \\ &= (A_0 \oplus A_1) \oplus (A_2 \oplus A_3) \\ &= \begin{cases} 1 & \text{Odd number of inputs is odd} \\ 0 & \text{Otherwise (including all 0 input)} \end{cases} \end{aligned}$$

Half adder

- Two single bit inputs
- Two outputs, named the Sum and Carry
- Not a "full adder" since it does not have a "carry" input

In		Out	
A	B	S	Cy
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Circuit for half adder:



Full adder

- Three single bit inputs: A, B and "carry in", Cy_i
- Two single bit outputs: sum, S, and "carry out", Cy_o

$$\begin{aligned} S &= Cy_i \oplus A \oplus B \\ Cy_0 &= AB + Cy_i B + Cy_i A \end{aligned}$$

Two's complement

Get the two's complement for a n-bit binary

1. Negate each bit (One's complement)
2. add 1 to the above result

Binary subtraction

- A minus B is the same as A plus the twos complement of B
- If $A > B$, B minus A will yield the two's complement of A minus B

Ten's complement

Get the ten's complement for a n-digit decimal number:

1. For each digit, find the number needed to add to that digit to get 9 (finding nine's complement)
2. Add $(1)_{10}$ to the nine's complement

BCD subtraction

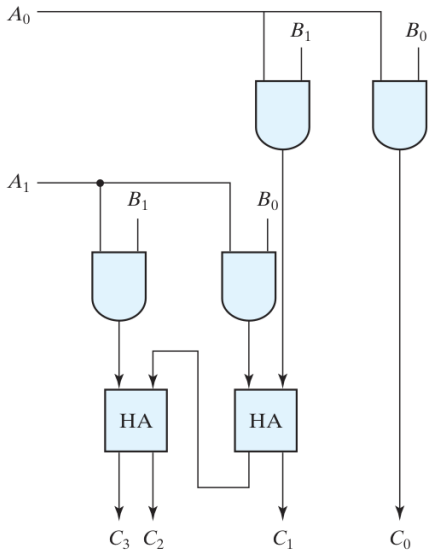
Subtracting two decimal numbers is the same as adding one number to the ten's complement of the other.

Binary multiplier

Multiplication in binary is no different from multiplication in decimal. Multiplying two binary digits is to compute the AND of the two digits.

$$\begin{array}{cccc} & & B_1 & B_0 \\ & & A_1 & A_0 \\ \hline & & A_0B_1 & A_0B_0 \\ & A_1B_1 & A_1B_0 & \\ \hline C_3 & C_2 & C_1 & C_0 \end{array}$$

We can express this using the logic ciruit:



where HA represent a half adder circuit.

Magnitude comparator

Given two n bit binaries, A, B, a *magnitude comparator* outputs three bits, representing whether $A > B$, $A = B$, $A < B$.

Equality: we can check the equality of two 1-bit binaries using XNOR:

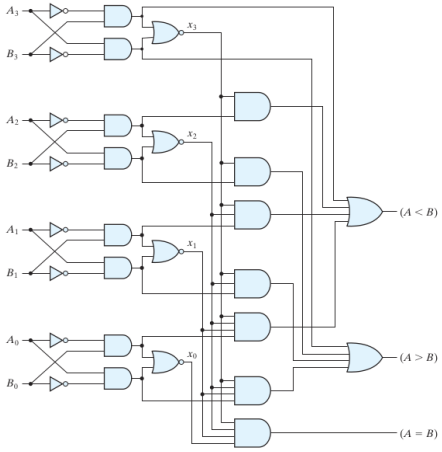
$$\begin{aligned} x_i &= A_i B_i + A_i' B_i' \qquad\qquad i = 0, 1, \ldots, n \\ &= (A_i \oplus B_i)' \end{aligned}$$

x_i is 1 whenever $A_i = B_i$.
Two n-bit binaries are equal only when all of their bits are equal.

$$(A = B) = \prod_{i=0}^{n} (A_i \oplus B_i)' = \left(\sum_{i=0}^{s} A_i \oplus B_i \right)'$$

If A and B were 4 bits:

$$\begin{aligned} (A > B) &= A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0' \\ (A < B) &= A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1 + x_3 x_2 x_1 A_0' B_0 \end{aligned}$$



Decoders, encoders, MUX, ALU

From n bits, we can make 2^n combinations. We can construct 2^n minterms/maxterms from n inputs.

Decoder

A *decoder* converts n binary inputs into at most 2^n unique binary outputs.

- each output present a unique minterm of the n inputs

Truth table for non-complemented minterms:

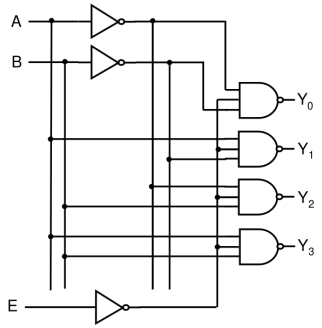
Inputs		Outputs			
B	A	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- Y_0 represents the minterm AB
- only a single output equal to 1 at all times
- constructed using AND gates

Truth table for complemented minterms:

Inputs			Outputs			
B	A	E	Y_0	Y_1	Y_2	Y_3
0	0	0	0	1	1	1
0	1	0	1	0	1	1
1	0	0	1	1	0	1
1	1	0	1	1	1	0
X	X	1	1	1	1	1

- The first column is $Y_0' = A'B'$, or $Y_0 = (A + B)$
- only a single output equal to 0 at all times
- constructed using NAND gates
- added ENABLE pin to take decode in/out of action

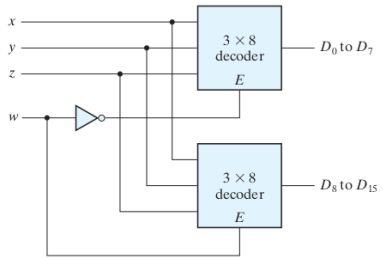


For a specific input combination, only the corresponding output will be 0, which equals E. When $E = 1$, that output changes from 0 to 1.

Decoders and decmultiplexers: Decoders with ENABLE function as *demultiplexers* (also called a *decoder-demultiplexer*).

Demultiplexers receive a line of data, and directs it to one of many possible output lines.

By connecting two 3-to-8 line decoders, we can make a 4-to-16 line decoder.



We can construct logic functions by connecting decoder output to logic gates.

If the decoder uses NAND gates:

- Implement ORs in the minterm using NAND gates (two level NAND gate is the same as two level AND-OR)

If the decoder uses AND gates:

- Connect the outputs corresponding to the terms in the minterms of the logic function using OR gates
- Connect outputs to a NOR gate for all outputs not a part of the minterm of the logic function (We are constructing minterms for Y', then complementing Y' to get Y)

Encoder

An *encoder* performs the inverse operation of a decoder. It has at most 2^n inputs and n outputs.

- When the encoder is "active low", then only a single input may be 0 at all times;
- Otherwise, only a single input may be 1 at all times;
- input of all 0s or all 1s is invalid

A priority encoder:

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

- D_3 has the highest priority, since x and y are 11 as long as D_3 is 1, no matter D_0, D_1, D_2
- A valid output is indicated by $V = 1$
- Input of all 0 sets V to be invalid

Multiplexers

A *multiplexer* (MUX) directs one of 2^n information lines to a single output line. The selection is determined by an additional n selection line.

Truth table for a 4 data line MUX with 2 selection lines:

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

We can write the minterms:

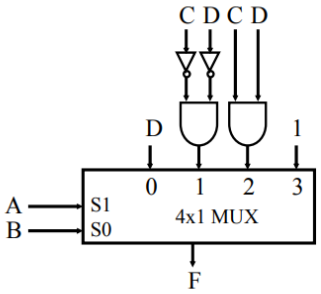
$$Y = (S_1'S_0'I_0) + (S_1'S_0I_1) + (S_1S_0'I_2) + (S_1S_0I_3)$$

We can use multiplexers to implement logic functions.

- The selection lines are inputs
- The logic 1 on a dataline selects that minterm

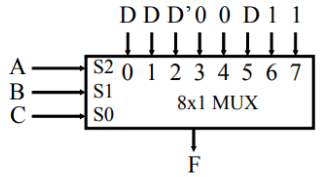
4 Variable function using 4-1 MUX: Use the selection lines as the third and fourth input

A	B	C	D	F	
0	0	0	0	0	F=D
0	0	0	1	1	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	1	F=C'D'
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	0	F=CD
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	1	
1	1	0	0	1	F=1
1	1	0	1	1	
1	1	1	0	1	
1	1	1	1	1	



4 Variable function using 8-1 MUX: Use the selection lines as the fourth input.

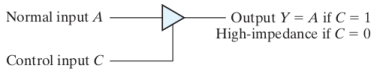
A	B	C	D	F	
0	0	0	0	0	F=D
0	0	0	1	1	
0	0	1	0	0	F=D
0	0	1	1	1	
0	1	0	0	1	F=D'
0	1	0	1	0	
0	1	1	0	0	F=0
0	1	1	1	0	
1	0	0	0	0	F=0
1	0	0	1	0	
1	0	1	0	0	F=D
1	0	1	1	1	
1	1	0	0	1	F=1
1	1	0	1	1	
1	1	1	0	1	F=1
1	1	1	1	1	



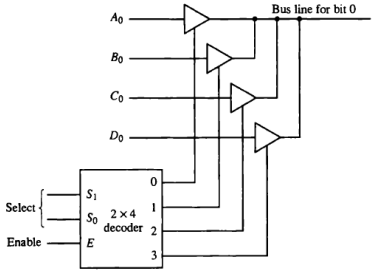
Three state gates

The output of a three state gate can be:

1. Logic 1
2. Logic 0
3. High impedance: output behaves like an open circuit



We can connect the outputs of three state gates together, as long as all gates are in high impedance mode, with the exception of one.



Arithmetic logic unit

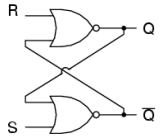
Consists of

- Two n-bit inputs
- m-bit instruction selection input
- n-bit output

Flip-flops and Registers

Set-reset latch

NOR: $S = 1, R = 0$ sets $Q = 1, Q' = 0$. $S = 0, R = 1$ resets the latch to $Q = 0, Q' = 1$. $S = R = 1$ is INVALID, since both $Q, Q' = 0$.

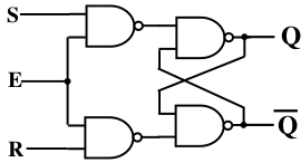


S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

NAND: Complement the inputs of the NOR S-R latch truth table to get the truth table for NAND S-R latch.

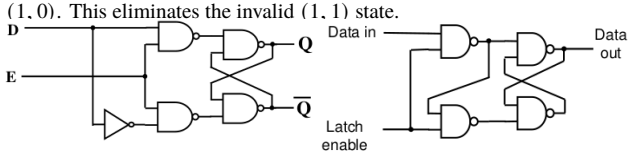
Gated SR latch

When $E = 1$, the circuit becomes an S-R NOR latch. When $E = 0$, the circuit is disabled, and Q, Q' latches onto their previous state.



Gated D latch

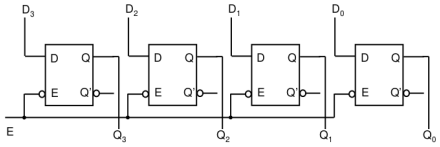
The inputs to the two NAND gates at stage 1 will always be $(0, 1)$ or $(1, 0)$. This eliminates the invalid $(1, 1)$ state.



E	S	R	Q	Q'
1	1	0	1	0
1	0	1	0	1
0	x	x	Q(t-1)	Q'(t-1)

Multi-bit D latch

By connecting the enables to a common line, we get a multi-bit latch.



Output enable

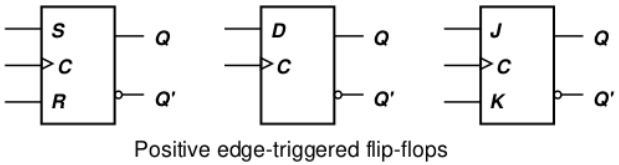
- Additional output enable input, R
- AND gate connects R and output of D latch, Q
- When $R = 1$, $Q = 1$, AND gate is 1, passing the value of Q
- When $R = 1$, $Q = 9$, AND gate is 0, passing the value of Q

Limitations of latch circuits

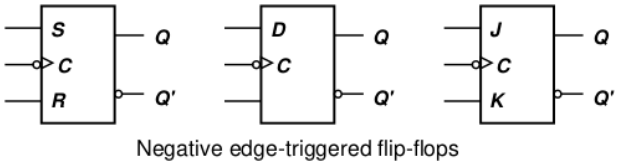
- When enable is active, the output immediately changes with the input
- Outputs can take on unintended states

Flip-flops

- Is a latch with a clock signal
- Ouputs only change on clock edges



Positive edge-triggered flip-flops

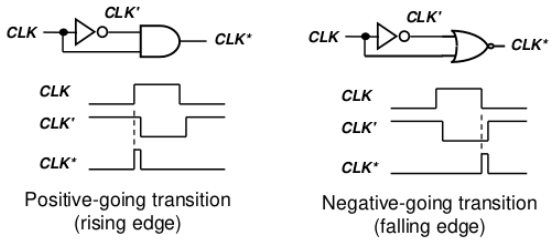


Negative edge-triggered flip-flops

A clock edge detector connects to the enable pin.

Clock edge detectors

Some detectors use propagation delay of circuits to generate a narrow ouput pulse.



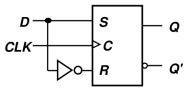
SR Flip-flop

A gated SR latch with a clock edge detector at the enable.

D Flip-flop

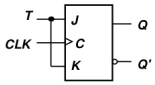
Made by ensuring only one of S or R will be logic 1 at all times. Removing the LATCH and INVALID state.

Made with JK F-F by moving the LATCH and TOGGLE state.



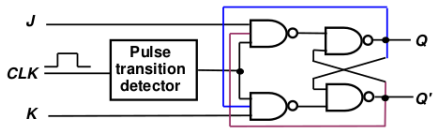
T Flip-flop

T connects J and K together, removing the set-reset states of the JK flip-flop. Only toggle and no change is possible.



JK Flip-flop

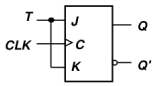
INVALID state becomes TOGGLE that complements the current output.



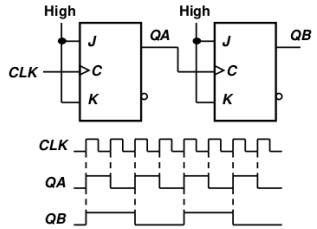
J	K	CLK	Q(t+1)	Comments
0	0	↑	Q(t)	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	Q(t)'	Toggle

T Flip-flop

T connects J and K together, removing the set-reset states of the JK flip-flop. Only toggle and no change is possible.

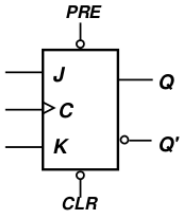


Frequency division: Using toggle, T flip-flops can divide frequencies by 2.



Asynchronous inputs

- Flip-flops are synchronous devices, since the output can only changes for every clock edge.
- Asynchronous flip-flops add **CLEAR,PRESET** pins
- Q = 1 whenever PRESET= 1
- Q = 0 whenever CLR= 1



Counters

- Synchronous counters use the same clock for all flip-flops. All outputs change at the same time
- Asynchronous counters allow outputs of flip-flops to be used as clock for other flip-flops.
- Using n flip-flops, our counters can have at most 2^n states

Binary ripple UP counters

- Are constructed the same way as frequency divisors
- Q is connected to CLK of the next flip-flop
- Propagation delay will cause invalid counts

Ripple counters with $< 2^n$ states: Using T flip-flops with a CLEAR pin. Add a circuit that activates CLEAR when ever a particular state is reached.

- Ripple DOWN counters are made by connecting Q' to CLK of the next flip-flop

Sychronous counters

Design with JK F-Fs

1. Draw state diagram
2. Draw state table
3. Simplify logic

For every current state, the state table lists the F-F inputs needed to switch from a current state to a specified next state.

Current state		Next state		Flip-flop inputs	
A_1	A_0	A_1^+	A_0^+	TA_1	TA_0
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

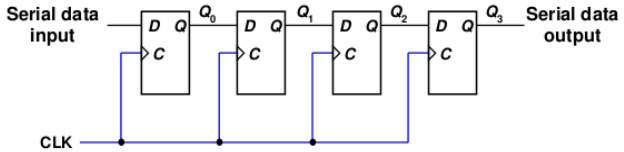
Design with D F-Fs

Since the output of the D F-F is exactly equal to the input whenever ENABLE is active, we only need current state, and next state in our state table.

Shift registers

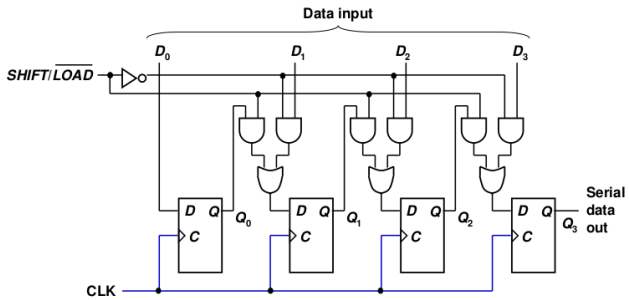
Is an arrangement of sychronous F-F used to move bits.

Serial in/ serial out: The bits move from F-F to F-F

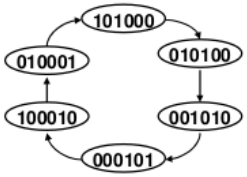


Serial in/ parallel out: Additional lines tap the outputs of each F-F.

Parallel in/ serial out: Additional 2-to-1 multiplexer circuits



Ring counter: Satisfy the state transition diagram



Johnson counter: Satisfy the state transition diagram