CH3 Boolean Algebra and Digital Logic

3.1 Intro

Two states, on or off, 0 or 1, false or true.

3.2 Boolean Algebra

I used this to check my work: https://web.stanford.edu/class/cs103/tools/truth-table-tool/

3.2.1 Boolean Expressions

Boolean variables (can be true=1 or false=0).
Boolean operations (AND, OR, NOT, etc).
Boolean expressions (combinations of variables and operations)
Boolean function (result of a Boolean expression).
Truth Table (shows result for all possible variable values)

Boolean Product **x AND y** (written xy in this book or x·y, x^y, x&y depending on style)

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Sum **x OR y** (written x+y in this book, also x∨y, x|y)

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

NOT x (written x’ in this book, also $\bar{x}$, ~x, ¬x, !x)

x	x’
0	1
1	0

x XOR y (written as x⊕y (type 2295 and press alt-x))

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

Functions:

For example, consider the function F = x'y + y'z, the easiest way is to create a truth table:

x	y	z	x'y	y'z	F
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	0	0	0
1	1	1	0	0	0

The 4th and 5th columns are intermediate calculations and are not technically part of the truth table.

3.2.2 Boolean Identities

Boolean expressions can be simplified by using Boolean identities. They have AND forms and OR forms.

Identity Law: 1x=x, 0+x=x

Null Law: 0x=0, 1+x=1

Idempotent Law: xx=x, x+x=x

Inverse Law: xx'=0, x+x'=1

Commutative Law: xy=yx, x+y=y+x

Associative Law: (xy)z=x(yz), (x+y)+z=x+(y+z)

Distributive Law: x+(yz)=(x+y)(x+z), x(y+z)=xy+xz

Absorption Law: x(x+y)=x, x+xy=x

DeMorgan's Law: (xy)'=x'+y', (x+y)'=x'y'

Double Complement Law: x''=x

The easiest way to confirm these identities is to write out truth tables for each side.

3.2.3 Simplification of Boolean Expressions

Consider: (x+y)(xz+xz')+xy+y

```
(x+y)(xz+xz')+xy+y
(x+y)(x(z+z'))+xy+y          distributive law
(x+y)(x1)+xy+y               inverse law
```

```
(x+y)x+xy+y                identity law
xx+yx+xy+y                 distributive law
x+yx+xy+y                  idempotent law
x+xy+xy+y                  commutative law
x+xy+y                     idempotent law
x(1+y)+y                   distributive law
x1+y                       null law
x+y                        identity law
```

That was an easy one. Sometimes it is a lot more work. You'll study this in much more detail in PH220, so I'll move on.

3.2.4 Complements

DeMorgan's Theorem can be generalized to more variables: (xyz)'=x'+y'+z'

This makes it easy to find the complement of any Boolean expression. Simply change all variables to their complements and change all the ANDs to ORs and vice versa:

if
F=xy'+zx' (tt=FTFTTTFF)
Then
F'=(x'+y)(z'+x) (tt=TFTFFFTT)

Note we needed to add parens.

3.1.5 Representing Boolean Functions

There are two canonical forms: **sum-of-products form (SOP)** and **product-of-sums form (POS)**. Here's some examples:

F=xy+yz'+xyz - SOP

F=(x+y)(x+z')(y+z')(x+y) - POS

You can easily create the SOP form from the truth table:

```
x y z  F
0 0 0  0
0 0 1  1
0 1 0  1
0 1 1  1
1 0 0  0
1 0 1  1
1 1 0  0
1 1 1  0
```

Minterms are products that are negated for the 0's:

```
x y z  Minterm
0 0 0  x’y’z’
0 0 1  x’y’z
0 1 0  x’yz’
0 1 1  x’yz
1 0 0  xy’z’
1 0 1  xy’z
1 1 0  xyz’
1 1 1  xyz
```

To get the SOP term list the minterms that give you a 1 in the truth table:

F=x’y’z+x’yz’+x’yz+xy’z

Maxterms are sums that are negated for the 1’s:

```
x y z  Maxterm
0 0 0  x+y+z
0 0 1  x+y+z’
0 1 0  x+y’+z
0 1 1  x+y’+z’
1 0 0  x’+y+z
1 0 1  x’+y+z’
1 1 0  x’+y’+z
1 1 1  x’+y’+z’
```

To get the POS form list the maxterms that give you a 0 in the truth table:

F=(x+y+z)(x’+y+z)(x’+y’+z))(x’+y’+z’)

3.3 Logic Gates

Gates are used to make digital circuits.

3.3.1 Symbols for Logic Gates

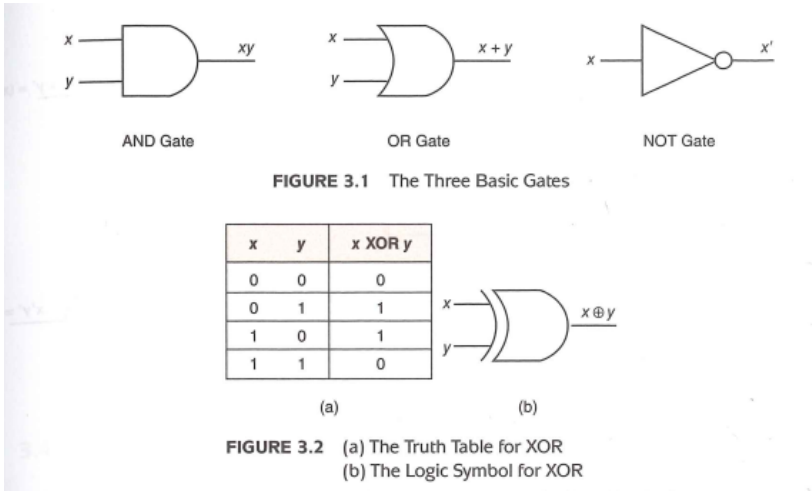


FIGURE 3.1 The Three Basic Gates

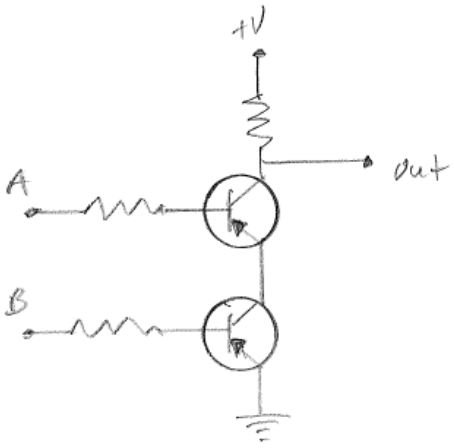
x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

(a) (b)

FIGURE 3.2 (a) The Truth Table for XOR
(b) The Logic Symbol for XOR

3.3.2 Universal Gates

Here's what a NAND gate looks like in transistors:



The inversion of the output is natural because of how transistors work.

The NAND gate is considered a universal gate because other gates can be constructed using them.

x	y	x NAND y
0	0	1
0	1	1
1	0	1
1	1	0

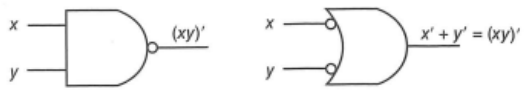


FIGURE 3.3 Truth Table and Logic Symbols for NAND

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

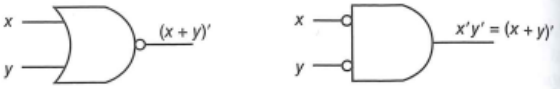


FIGURE 3.4 Truth Table and Logic Symbols for NOR

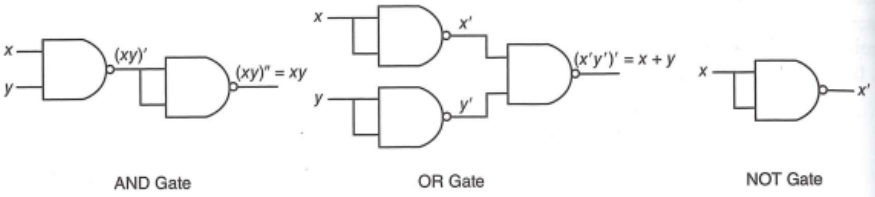


FIGURE 3.5 Three Circuits Constructed Using Only NAND Gates

Note the circles at the inputs and outputs, which mean to negate the voltage.

3.3.3 Multiple Input/Output Gates

Gates can have more than 2 inputs and can have two outputs (Q and Q').

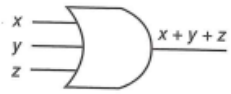


FIGURE 3.6 A Three-Input OR Gate Representing $x + y + z$

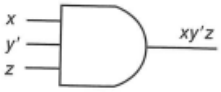


FIGURE 3.7 A Three-Input AND Gate Representing $xy'z$

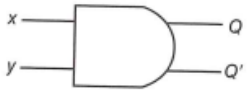


FIGURE 3.8 AND Gate with Two Inputs and Two Outputs

3.4 Karnaugh Maps (K-Maps)

Karnaugh Maps can simplify Boolean expressions into their simplest forms. This can save on transistors in complex circuits.

While this is an important topic, I'm not going to cover it in this class.

3.5 Digital Components

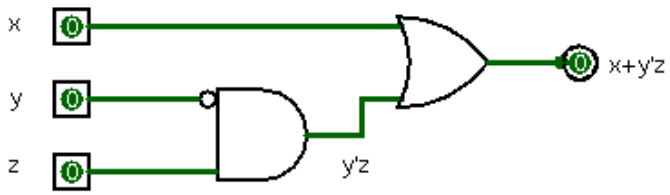
3.5.1 Digital Circuits and their Relationship to Boolean Algebra

Go here and download logisim: https://sourceforge.net/projects/circuit/files/latest/download

You may be able to double-click to run it. If not, go to the command line and type

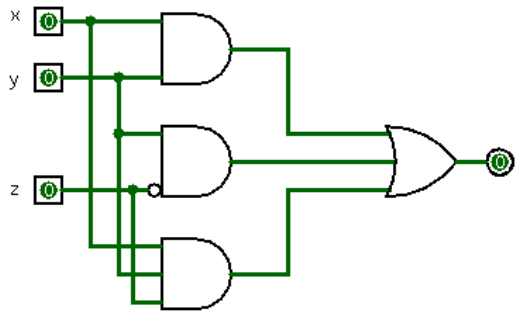
java -jar logisim-generic-2.7.1.jar

Here's the circuit in Fig 3.18 where F=x+y'z



Note the negation of the input for y.

Here's the circuit just below that one on page 161: F=xy+yz'+xyz



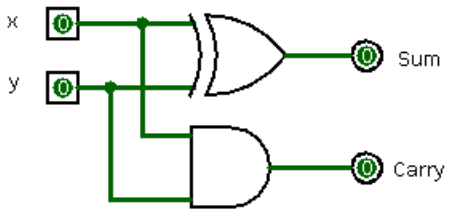
3.5.2 Integrated Circuits (IC's)

Prior to IC's computers were made with transistors on motherboards. With the advent of IC's the density of transistors could grow extremely fast. But gates are still the basic building blocks, it's just that there are a lot of them.

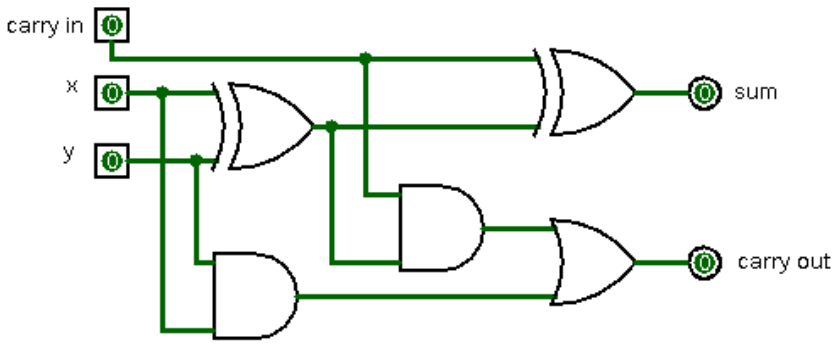
3.6 Combinational Circuits

These are circuits that don't have a memory of a previous state (like a flip-flop).

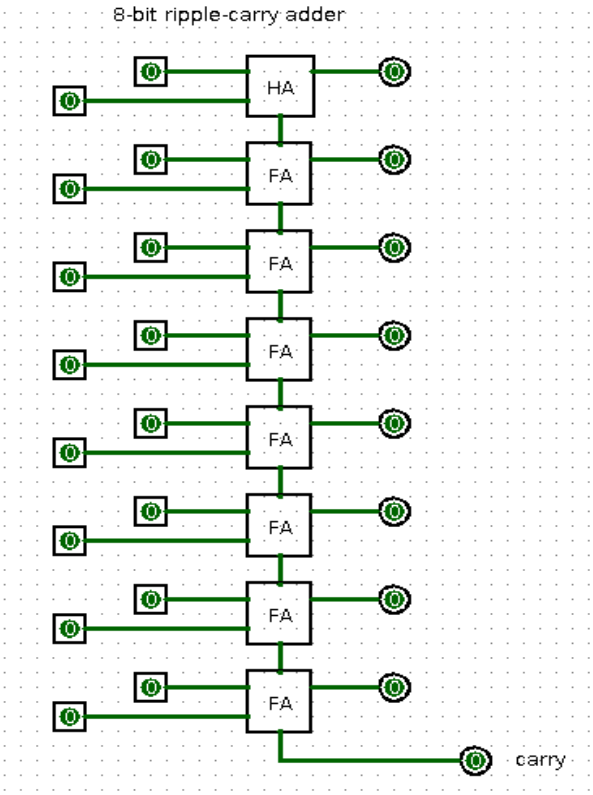
The Half-adder:



The Full adder:



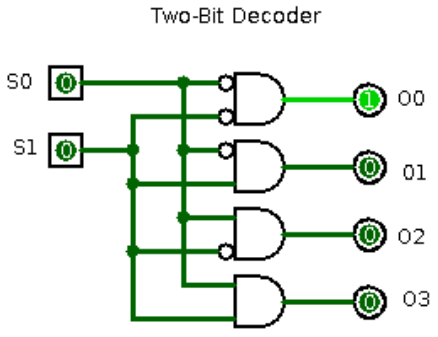
We can combine one half-adder and five full adders to make a 6-bit **ripple carry adder**:

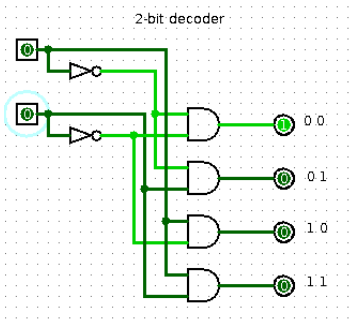


There are faster circuits for adding, but this one is much easier to understand. It will suffice for this class.

Another important circuit is the **decoder**. It takes N inputs and activates one of 2^N outputs. It is used to select bytes in RAM, decode machine language instructions, activate ALU circuits, etc.

Here's a simple 2-bit decoder (2-to-4 decoder):



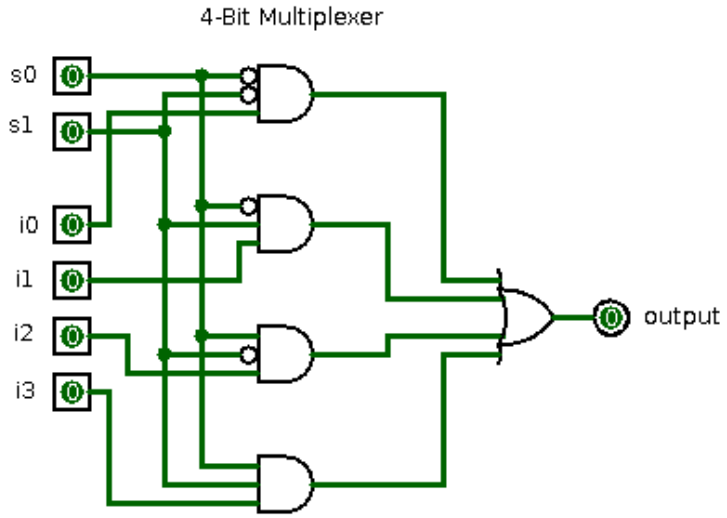


Note the positions of the circles on the AND gates. Can you visualize how this could be extended to 3 or more bits?

Sometimes the output of one circuit becomes the input of another circuit (a register may be connected to an adder, a multiplier, a bitshifter, etc. But only one input can be active at a time. The wires are still there, they're just temporarily blocked while some other input is connected.

The circuit to do this is called a **multiplexer**. There are N control lines that are selected using a decoder. There are 2^N inputs and one output. Which of the N inputs is connected to the output depends on the control lines.

Here's an example of a 4-bit multiplexer:



Note that O=i0 if s0=0 and s1=0. O=i1 if s0=0 and s1=1, etc.

Another useful circuit is a **parity generator**. Look at table 3.10 of the book:

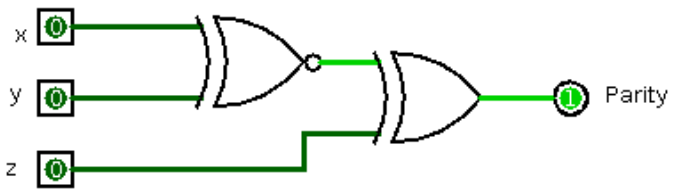
x	y	z	Parity Bit
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

TABLE 3.10 Parity Generator

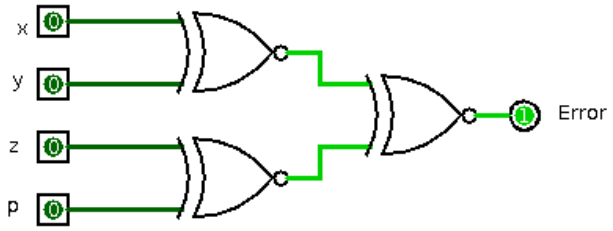
x	y	z	P	Error Detected?
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

TABLE 3.11 Parity Checker

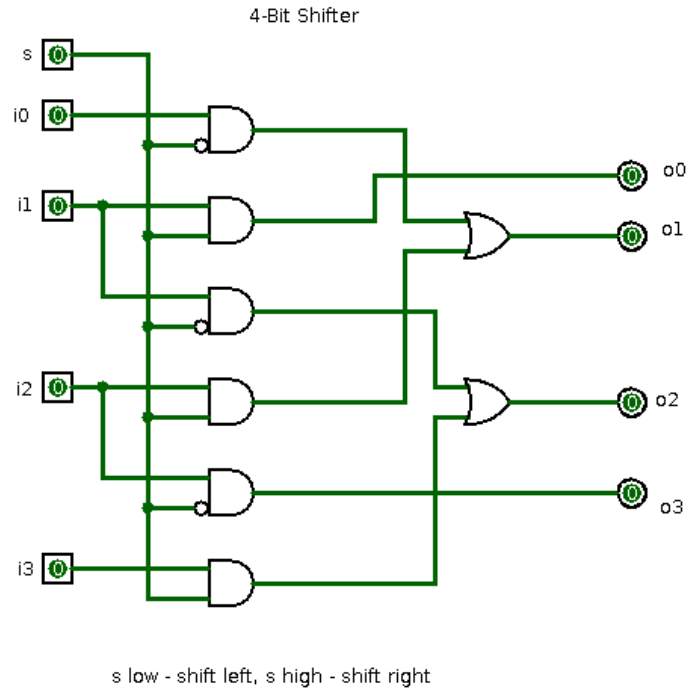
This scheme is called an odd parity generator. There are also even parity generators. Here's a circuit to generate odd parity for 3 bits:



Here's a parity checking circuit for odd parity:



Another circuit is the **bit shifter.** Here's one that will shift the inputs left if s=0 and right if s=1:



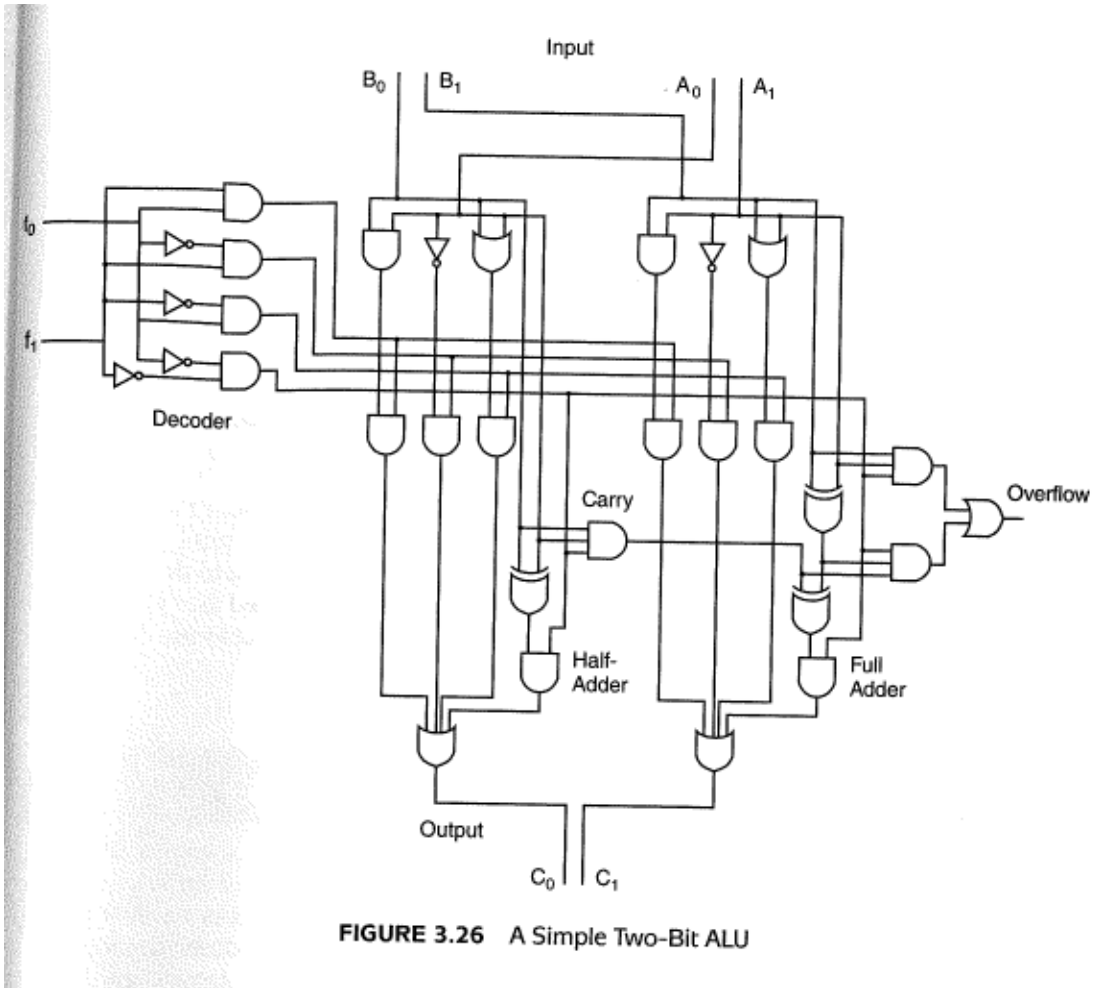


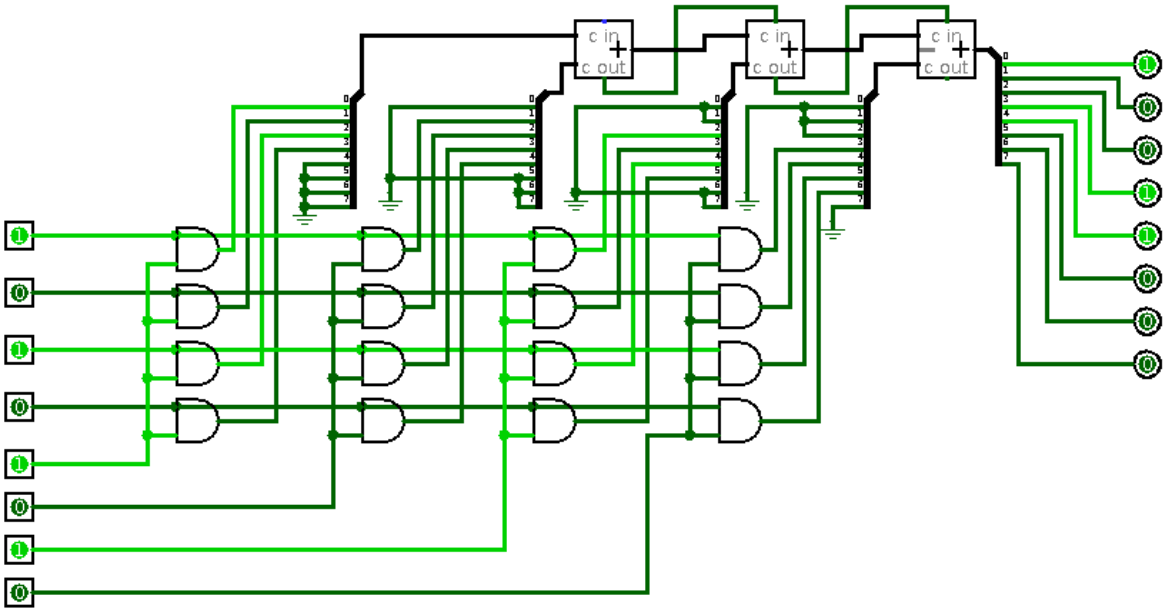
FIGURE 3.26 A Simple Two-Bit ALU

Note the decoder on the left. It chooses which operation to do. 00=addition, 01=NOT, 10=OR, 11=AND.

Consider multiplication. We repeat the multiplicand shifted left for each 1 in the multiplier. Thus multiplication is really addition.

```
    1101
    1011
  ------
    1101
   1101
  0000
 1101
--------
```

So can we use this idea in a circuit? The idea is to add the shifted multiplicand for every 1 in the multiplier.



4-bit multiplier (8-bit output).

3.7 Sequential circuits – circuits that have memory

3.7.2 Clocks – keep circuit parts synchronized. States need to be updated at a specific time. Combinational circuits need time to reach their final state.

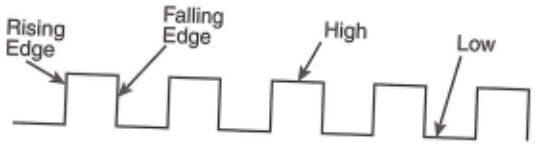


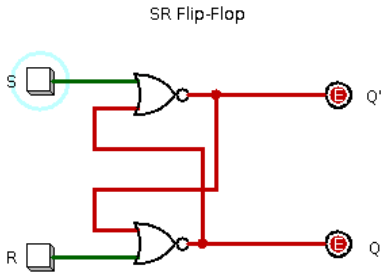
FIGURE 3.28 A Clock Signal Indicating Discrete Instances of Time

Note the timing of the clock pulses. There is a **rising edge**, a **falling edge** and two levels. Some circuits are **edge triggered** and some are **level triggered.**

3.7.3 Flip-flops – circuit that remains in a state until that state is changed.

Feedback – is important to ensure a stable state in a flip-flop.

The SR flip-flop (SR=Set/Reset)



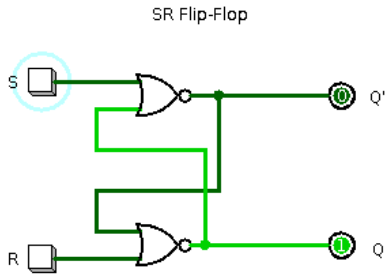
Initially it is in an unknown state. If S goes high it will go into a state where Q=1 and Q'=0.

Look at the TT for the NOR gate:

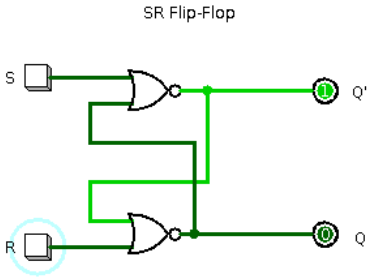
```
x y NOR
0 0  1
0 1  0
1 0  0
1 1  0
```

the only way you get a 1 at the output is if both inputs are 0. The feedback ensures that only one NOR gate will have a 1 output since the feedback will ensure the other one has a 0 output.

If S goes high it changes to

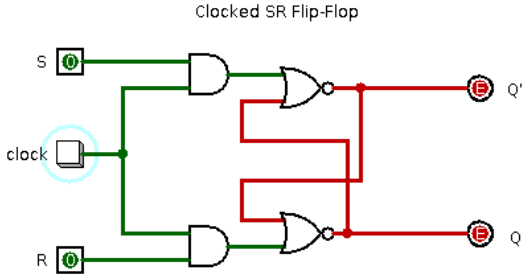


If R goes high it changes to



There's a problem with this circuit. If both S are R go high at the same time then both Q and Q' are 0. We must not allow this to happen.

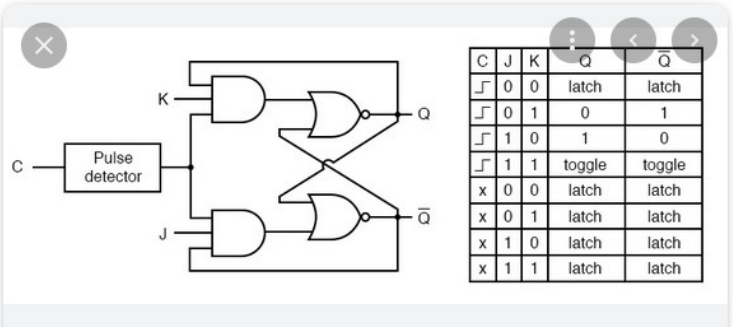
Often a clock is included, so S and R can vary but won't change the state until you are ready.



If S is high when the clock goes high, then it will set the state where Q=1. If R is high when the clock goes high then it will flip to Q=0.

If both S and R are high when the clock goes high it will go into the state where both Q and Q' are 0. But when the clock goes low it goes into a feedback loop with oscillation. This must be avoided.

The **JK flip-flop** introduces more feedback to eliminate the undefined state:

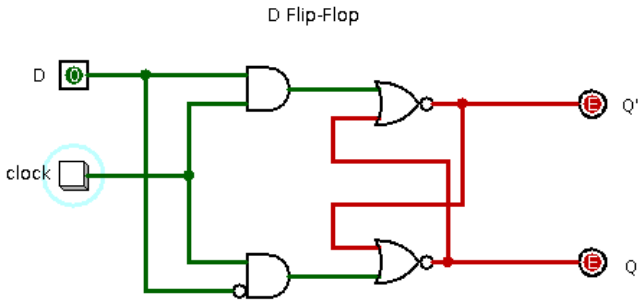


C	J	K	Q	$\overline{Q}$
⎍	0	0	latch	latch
⎍	0	1	0	1
⎍	1	0	1	0
⎍	1	1	toggle	toggle
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch

Since Q and Q' can never be 1 at the same time it prevents the state. Unfortunately I can't get this circuit to work in logisim. There's no way to get Q or Q' to 1.

If J=0, K=0 the JKFF will not change state. If J=0, K=1 it will reset (Q=0). If J=1, K=0 it will set (Q=1) and if J=1, K=1 it will toggle between the two states.

The most important flip-flop for computing is the **D flip-flop (**D=data). It sets the R to S':



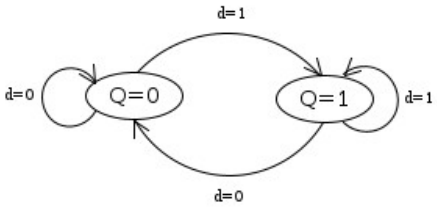
If D is high when the clock goes high, then Q=1. If D is low then Q=0.

This is the basic 1-bit memory circuit. Whatever the D is set to when the clock goes high is stored in the flip-flop until the clock is changed again.

3.7.4 Finite State Machines FSM's

FSMs can be used to show transitions between states in circuits. The states are represented as circles and the transitions are represented as arrows. It is assumed that a transition only takes place when the clock goes high.

For example, here is a FSM for a D flip-flop:



Note that if Q=0 then d=0 has no effect and if Q=1 then d=1 has no effect.

Algorithm State Machines are basically flowcharts. Here's one from the book:

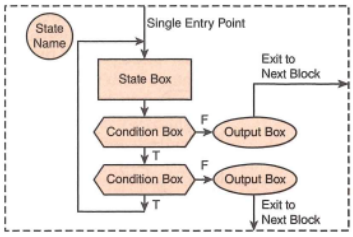


FIGURE 3.38 Components of an Algorithmic State Machine

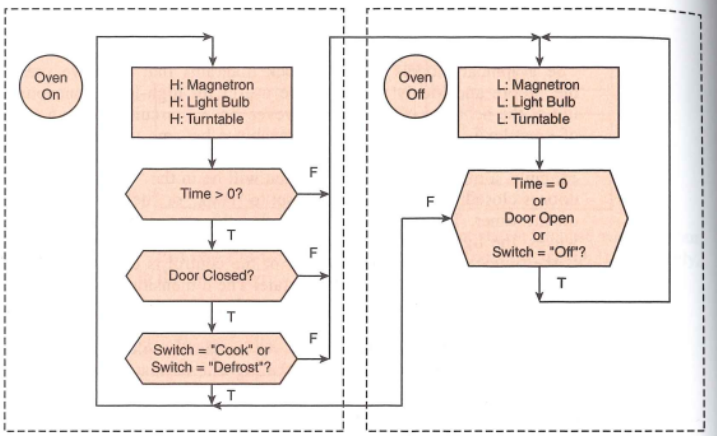
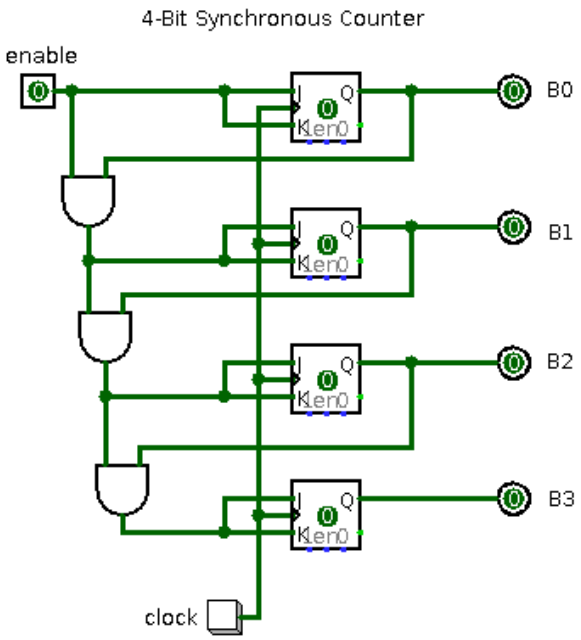


FIGURE 3.39 Algorithmic State Machine for a Microwave Oven

3.7.5 Examples of Sequential Circuits

Here's a 4-bit synchronous counter from the book (fig 3.41). When enable=1 the clock will count such that B=0000, 0001, 0010, etc.



Here's a 4x3 memory from the book (fig 3.42):

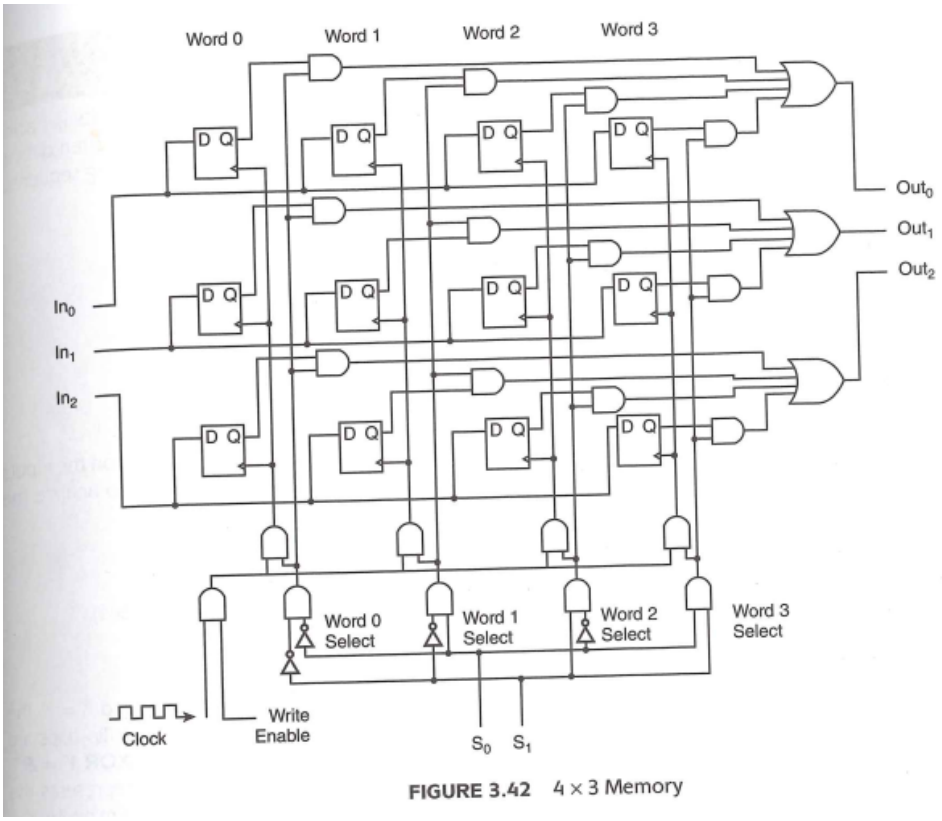


FIGURE 3.42 4 × 3 Memory