Logic Circuits

Robert Keller
November 2012

Gates

- Gates realize logic functions, e.g. electronically.
- Gates can be combined using what is essentially functional programming.
- We give symbolic versions of common logic functions.

and

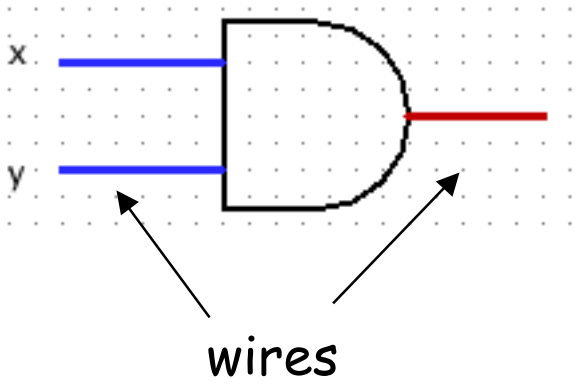
form 1 table:

x	y	and(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

arguments

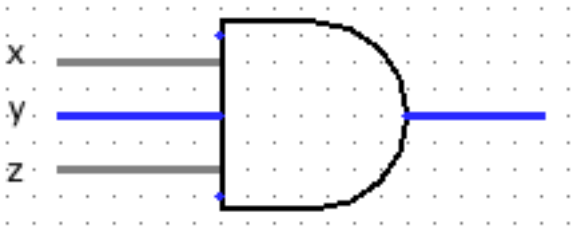
results

gate symbol

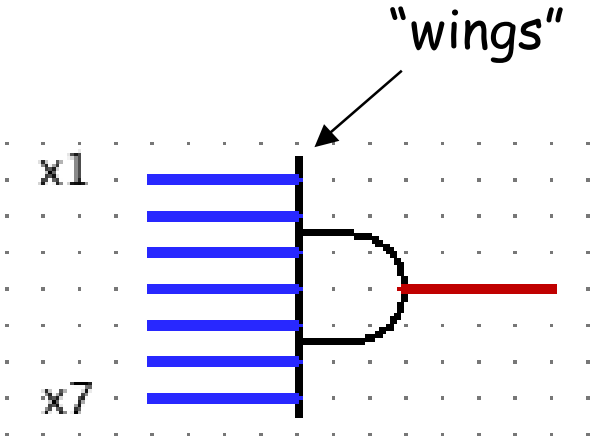


and with >2 inputs

- Because *and* is associative and commutative, the inputs can be regarded as a set with no particular order.



3 inputs



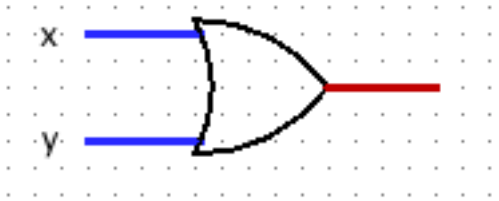
7 inputs

or

form 1 table:

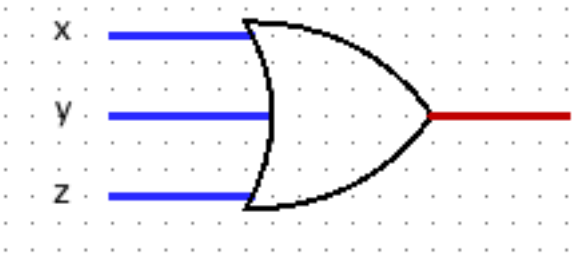
x	y	or(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

gate symbol

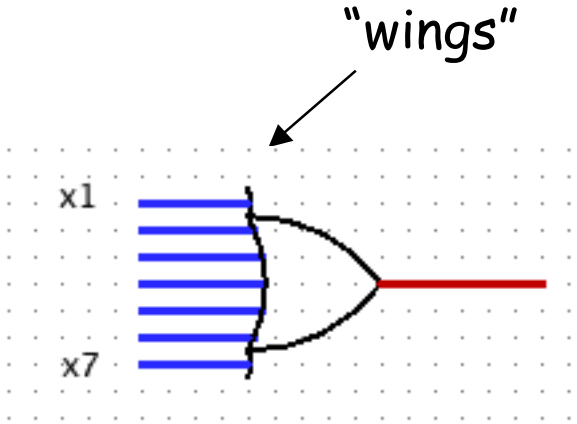


or with >2 inputs

- Because *or* is associative and commutative, the inputs can be regarded as a set with no particular order.



3 inputs



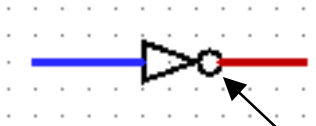
7 inputs

not (inverter)

form 1 table:

x	not(x)
0	1
1	0

gate symbol



inverter

"bubble" required

and, or, implies

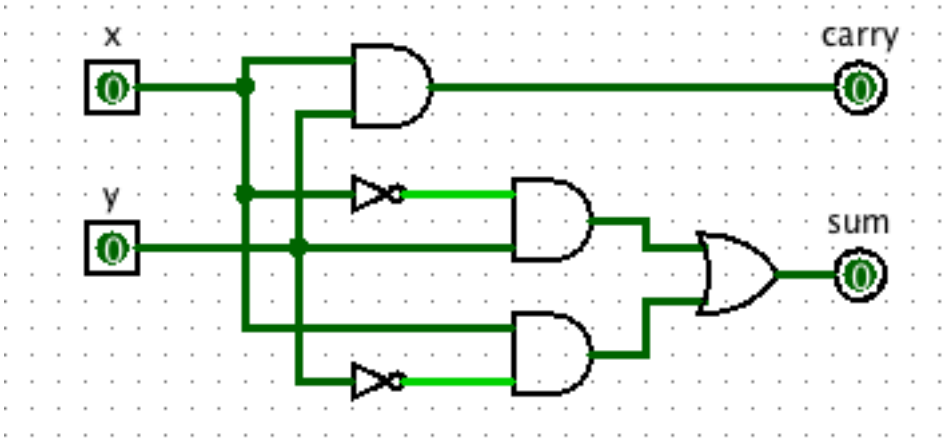
- 0 and x is
- 1 and x is

- 0 or x is
- 1 or x is

- 0 implies x is
- 1 implies x is

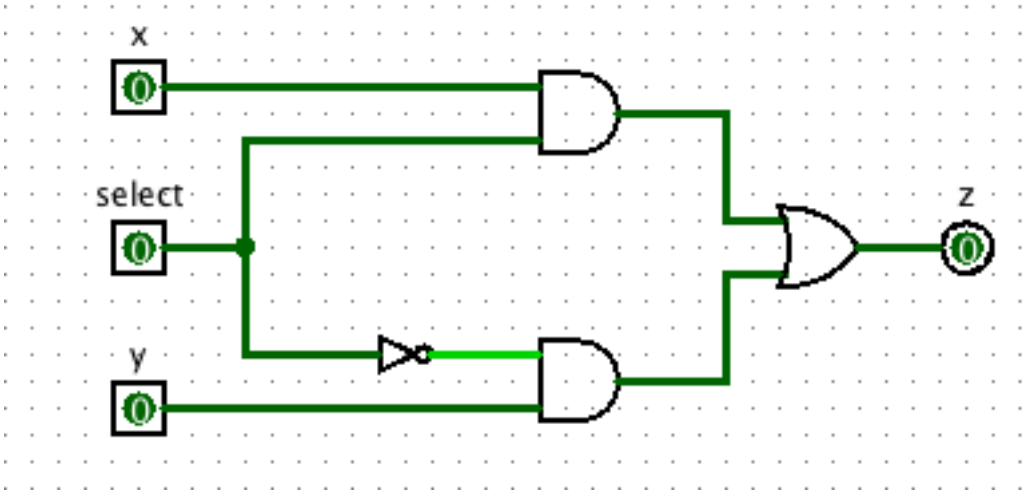
half-adder (HA) circuit

x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



1-bit multiplexer

- The gate equivalent of if-then-else
- z = select? x : y

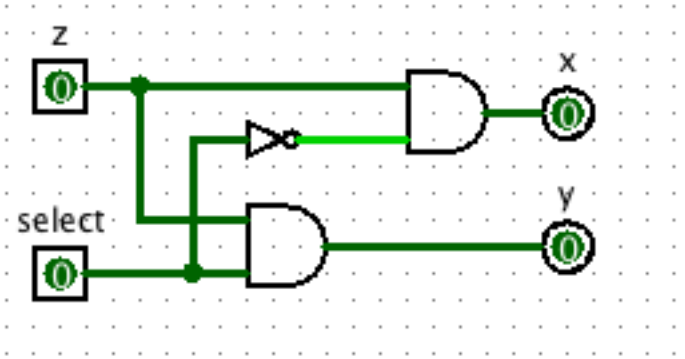


select	x	y	z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

1-bit demultiplexer

- The gate equivalent of if-then-else
- y = select? z : 0
  x = !select? z : 0

z	select	x	y
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

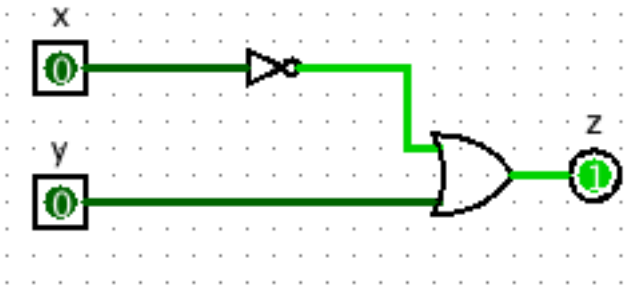


implies circuit

form 1 table:

x	y	implies(x, y)
0	0	1
0	1	1
1	0	0
1	1	1

z = implies(x, y)



Gate Repertoire

- By "repertoire" we mean a set of gate types, such as (2-input *and*, 2-input *or*, *inverter*).

Universal Repertoire

- A repertoire of n-input *and* and *or* gates, and *not*, is **adequate** to realize any switching function.

Proof that {and, or, not} are adequate

- Suppose f is an n-ary switching function.
- Basis: n = 1:
  - There are four 1-input functions.
  - $f_{00}(x) = 0 = \text{and}(x, \text{not}(x))$
  - $f_{01}(x) = x$
  - $f_{10}(x) = \text{not}(x)$
  - $f_{11}(x) = 1 = \text{or}(x, \text{not}(x))$
- (The subscripts of these functions are the result column of the truth table.)

x	$f_{00}(x)$
0	0
1	0

x	$f_{01}(x)$
0	0
1	1

x	$f_{10}(x)$
0	1
1	0

x	$f_{11}(x)$
0	1
1	1

Proof Continued: Induction Step

- Suppose that we can realize any n-ary function with {and, or, not}.
- Let f be an arbitrary (n+1)-ary function.
- Then $f(x_0, x_1, \ldots, x_n) =$
  $or(and(x_0, f(1, x_1, \ldots, x_n)),$
  $and(not(x_0), f(0, x_1, \ldots, x_n)))$
- But $f(0, x_1, \ldots, x_n)$ and $f(1, x_1, \ldots, x_n)$ are realizable, as they are n-ary functions.

Proof of the equality

- If $x_0 = 1$, the equality is:
  - $f(1, x_1, \ldots, x_n) =$
    $or(and(1, f(1, x_1, \ldots, x_n)),$
    $and(not(1), f(0, x_1, \ldots, x_n)))$
- But the RHS is
  $or(and(1, f(1, x_1, \ldots, x_n)), and(0, \_\_\_\_)))$

$= or(f(1, x_1, \ldots, x_n), 0)$

$= f(1, x_1, \ldots, x_n)$

so the equality checks for the case $x_0 = 1$.

Proof of the equality

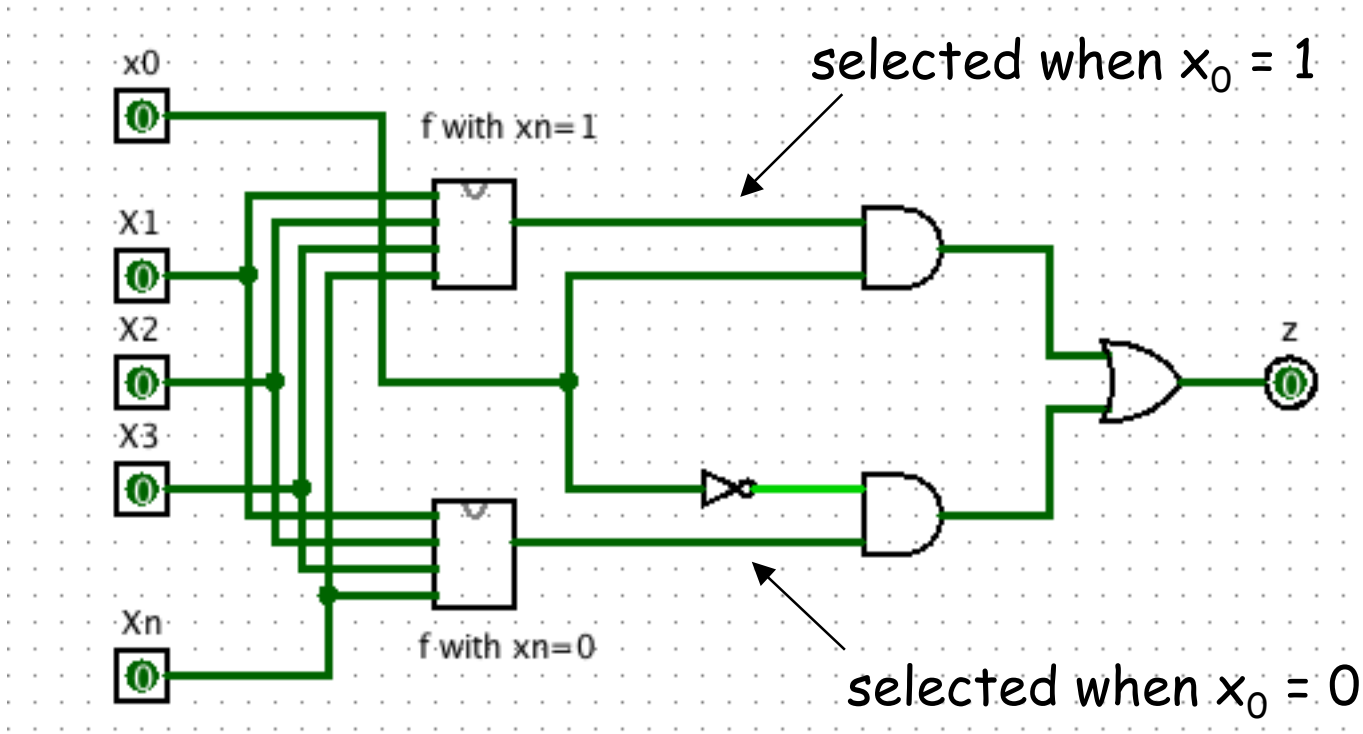
- If $x_0 = 0$, the equality is:
  - $f(0, x_1, \ldots, x_n) =$
    $or(and(0, \quad f(1, x_1, \ldots, x_n)),$
    $and(not(0), f(0, x_1, \ldots, x_n)))$
- But the RHS is
  $or(and(0, \_\_\_\_), and(1, f(0, x_1, \ldots, x_n))))$

$= or(0, f(0, x_1, \ldots, x_n))$

$= f(0, x_1, \ldots, x_n)$

so the equality checks for the case $x_0 = 0$.

Diagrammatic Proof



This is an application of the Boole-Shannon principle.

1-bit multiplexer

Proof as Synthesis

- The preceding proof determines one synthesis method for an arbitrary switching function:
  - If the function is of 1-ary, it is one of the four basis functions.
  - If the function is n+1-ary, then decompose it into a composition of n-ary functions, *and*, *or*, and *not*.

Synthesis Example

- Synthesis the function f represented by this truth-table:

w	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Synthesis Example

- Decompose on w

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

w=0

w	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

w=1

Synthesis Example

- Decompose each on x

y	z
0	0
1	1

x=0

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

w=0

y	z
0	1
1	1

x=1

w	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

y	z
0	1
1	1

x=0

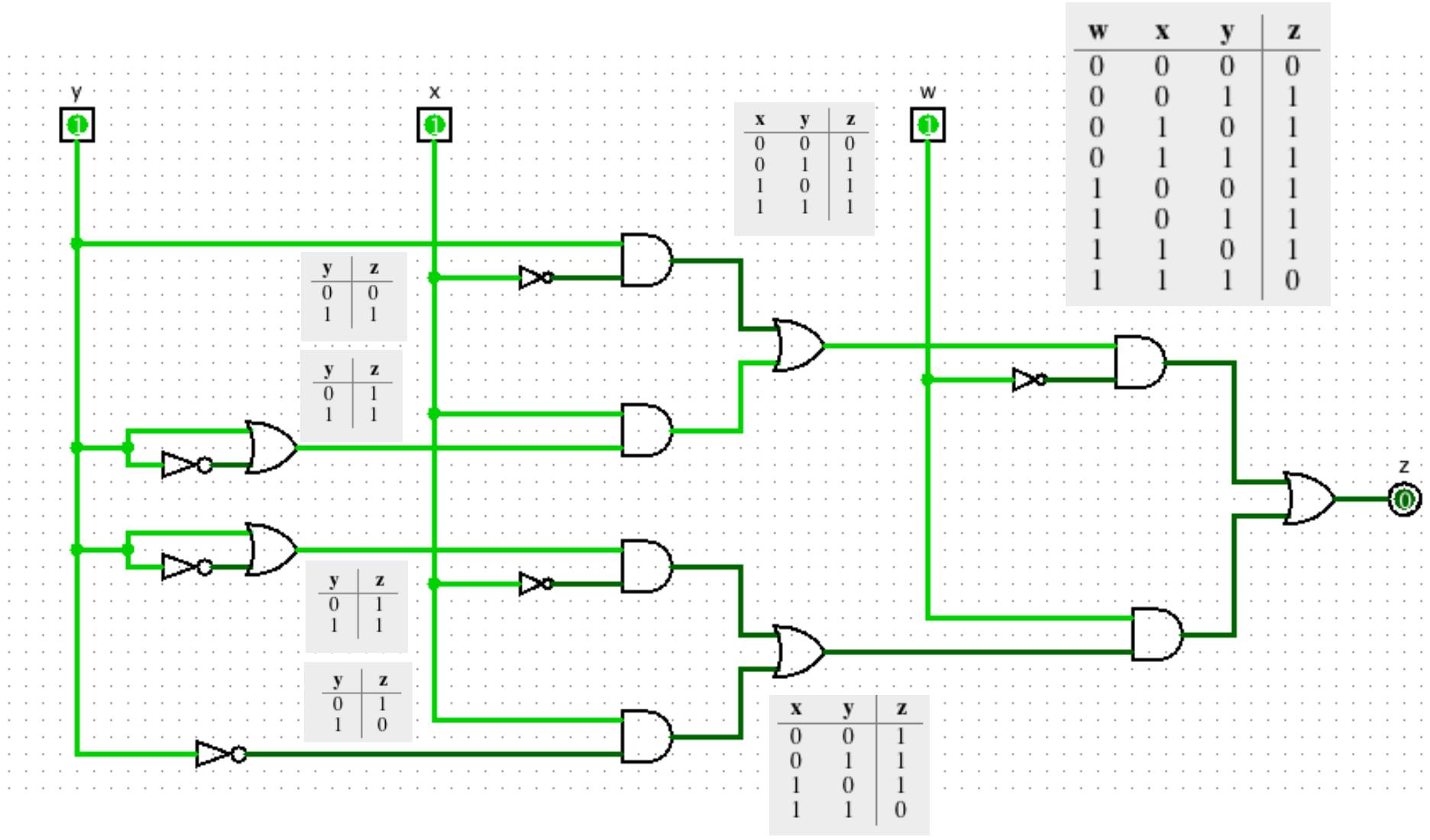
x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

w=1

y	z
0	1
1	0

x=1

Synthesis Example



Another Synthesis Method

- A problem with the previous method is that it tends to produce circuits with long paths (= delay).

- The next method reduces this problem.

Minterm Synthesis

- For a given set of variables, a minterm is a product (and) of those variables and their negations.
- Example: 3 variables: x, y, z
  - Some minterms: xyz', x'yz', …
- Each minterm represents a function that is "on" (1) for exactly one combination of variables
  - e.g. xyz' is on for 110, x'yz' is on for 010.

Minterm Representation

- Any function can be represented as the sum (*or*) of its minterms.

- These correspond to the rows of the truth table for which the function is 1.

Minterm Example

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Minterm representation is

w'xy' + w'xy + wx'y + wxy'

corresponding to rows

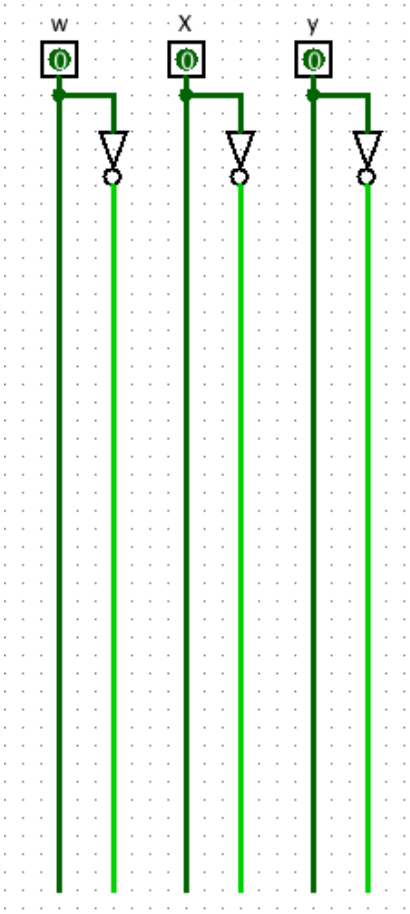
010 011 101 110

numbering

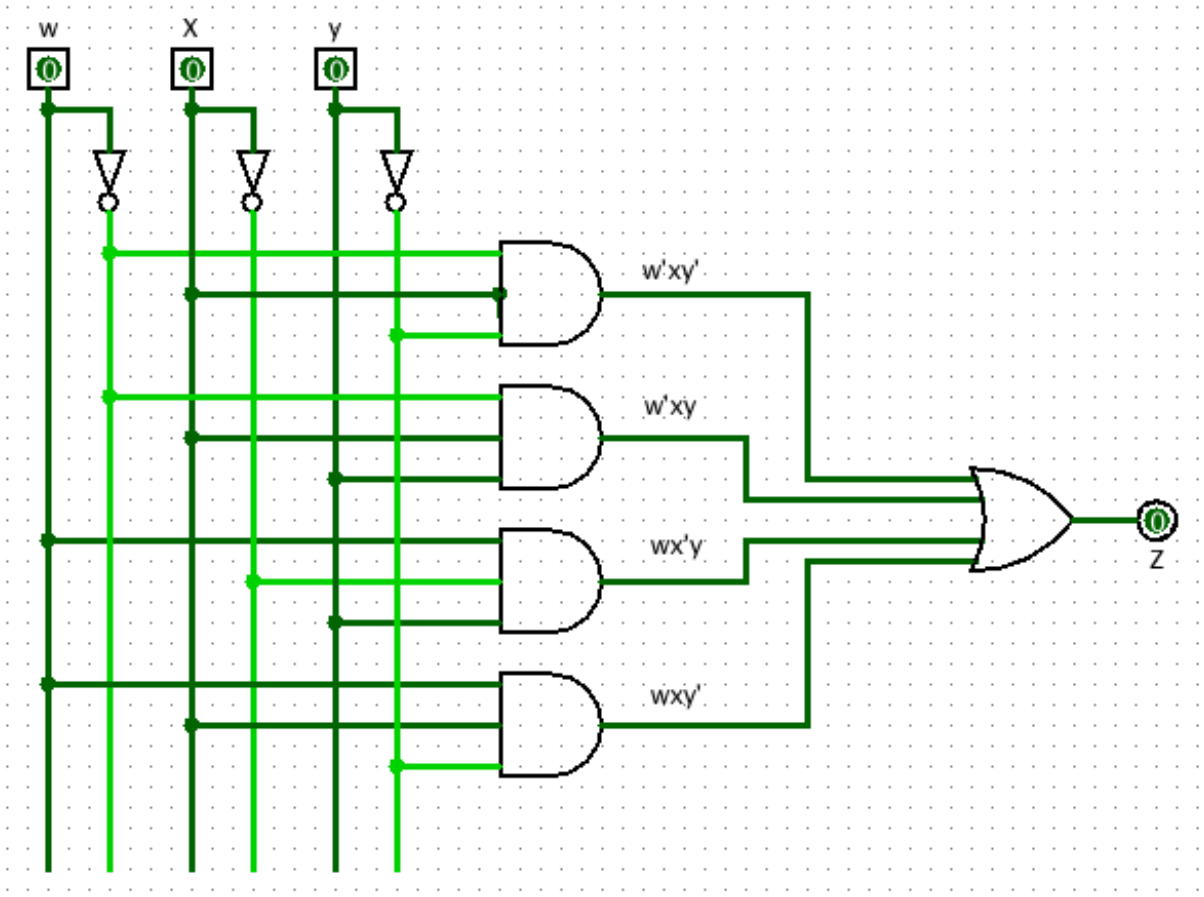
$\Sigma(2, 3, 5, 6)$

Minterm Synthesis

- Use "rails" for clarity

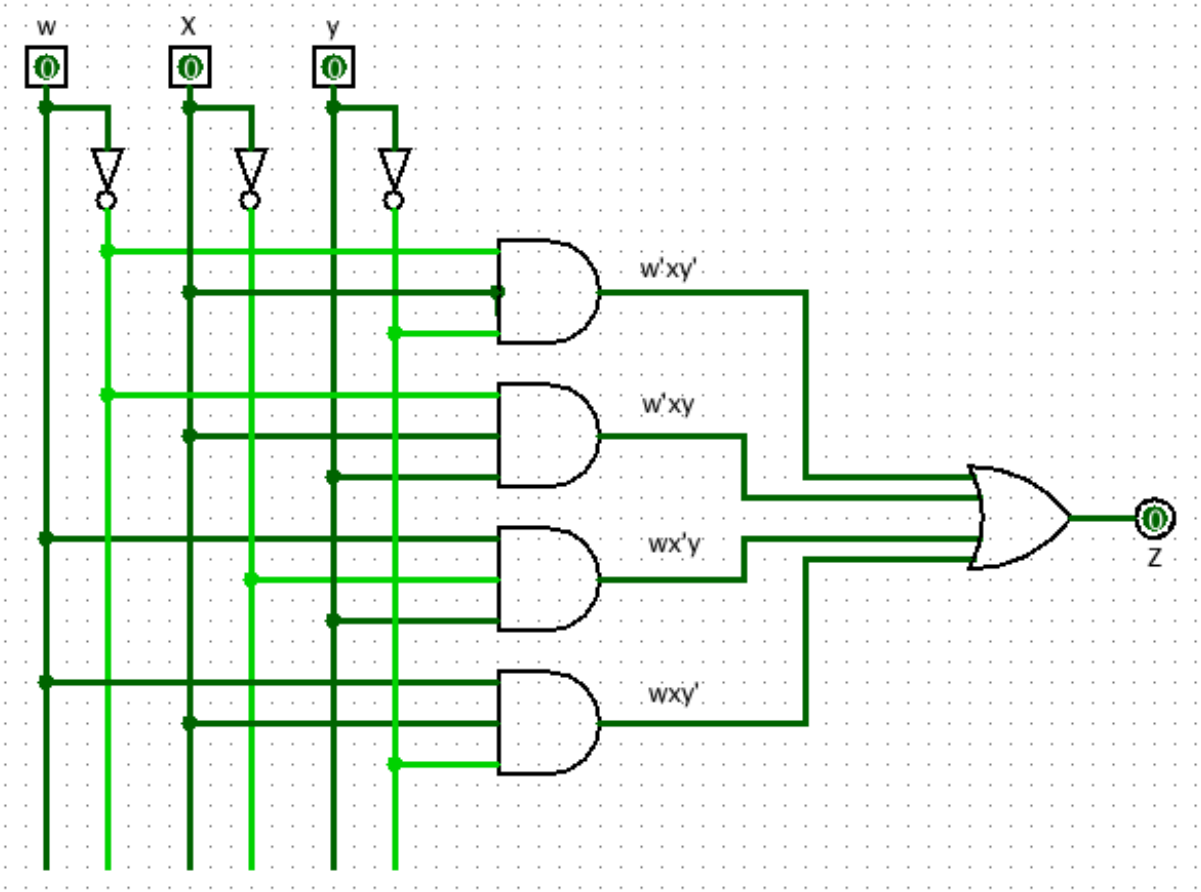


Minterm Synthesis



w'xy' + w'xy + wx'y + wxy'

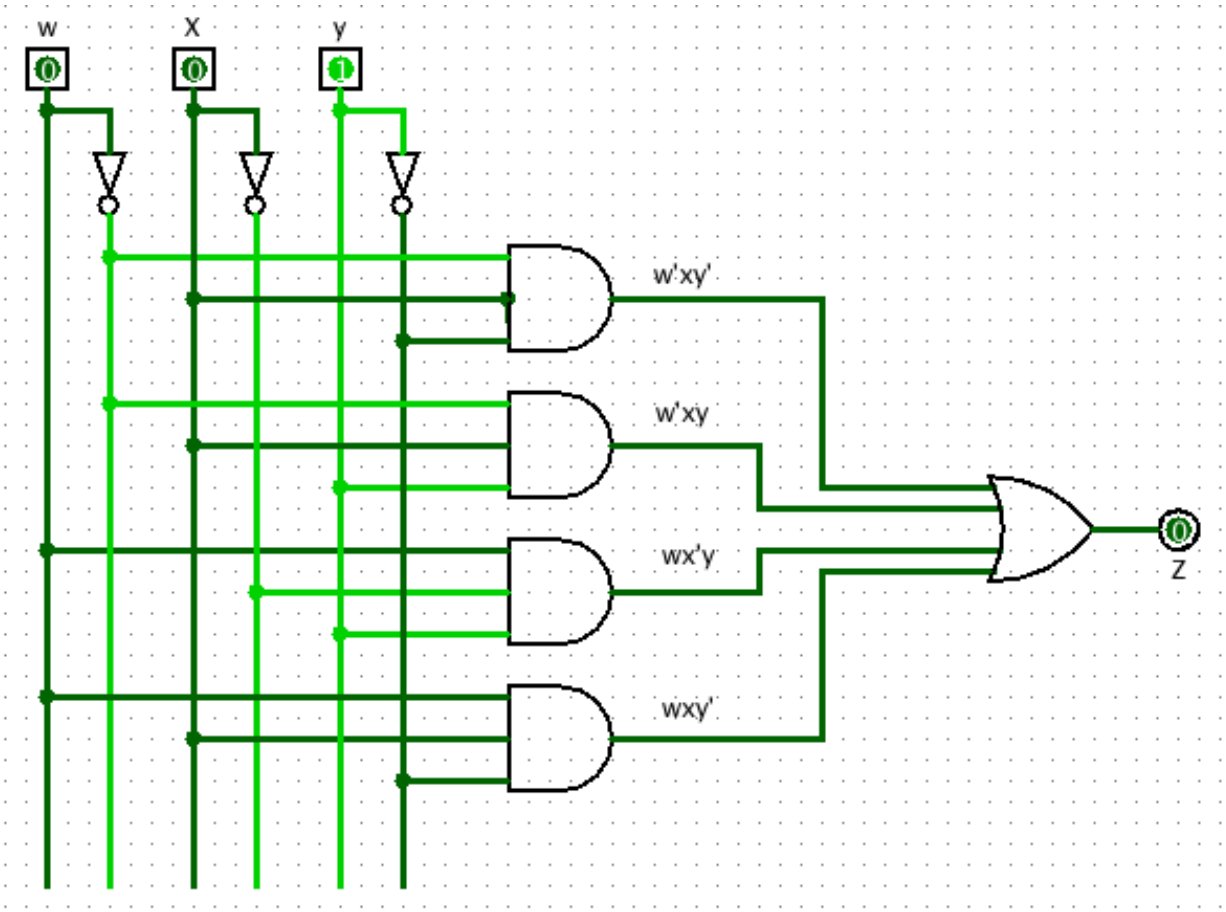
Illustrating Combinations



w'x'y'

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

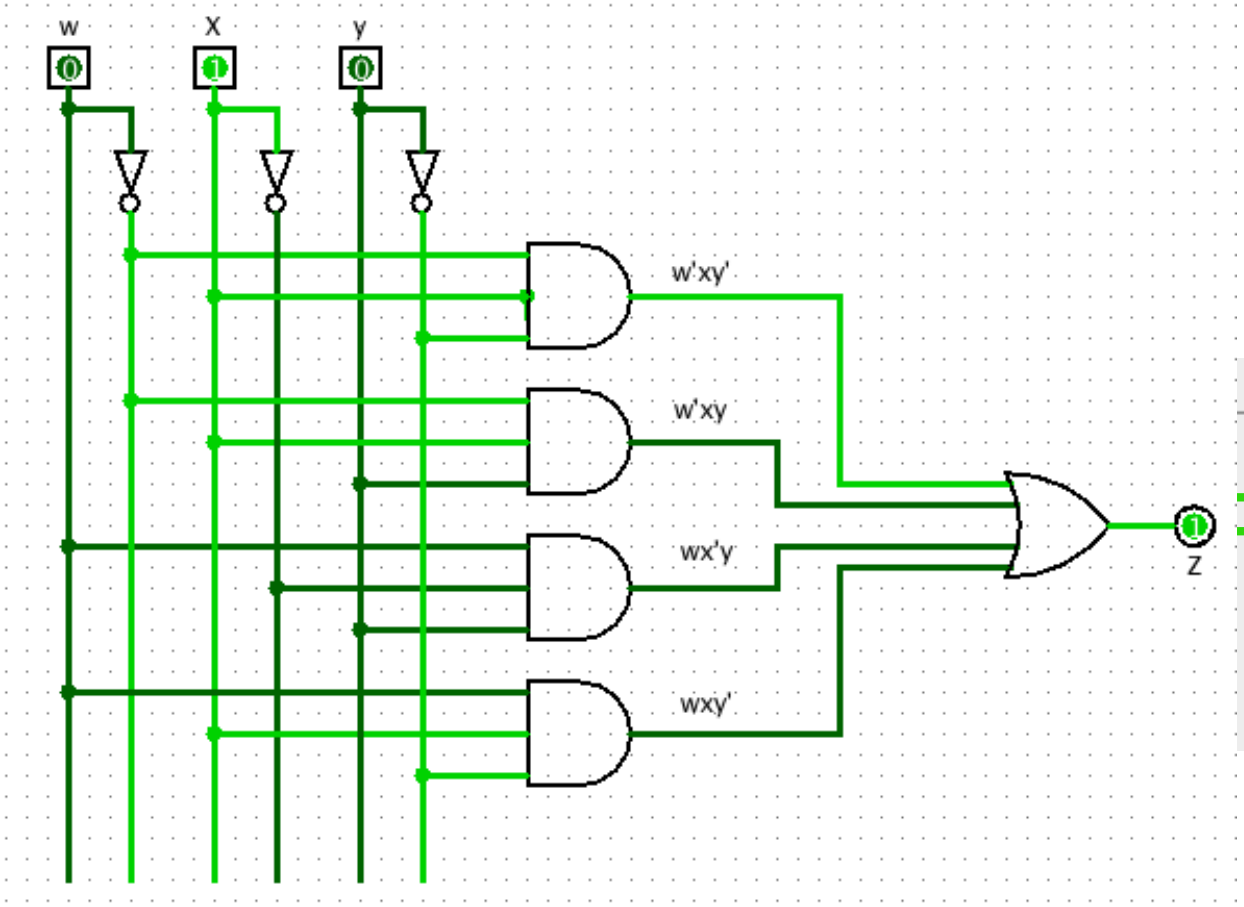
Illustrating Combinations



w'x'y

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

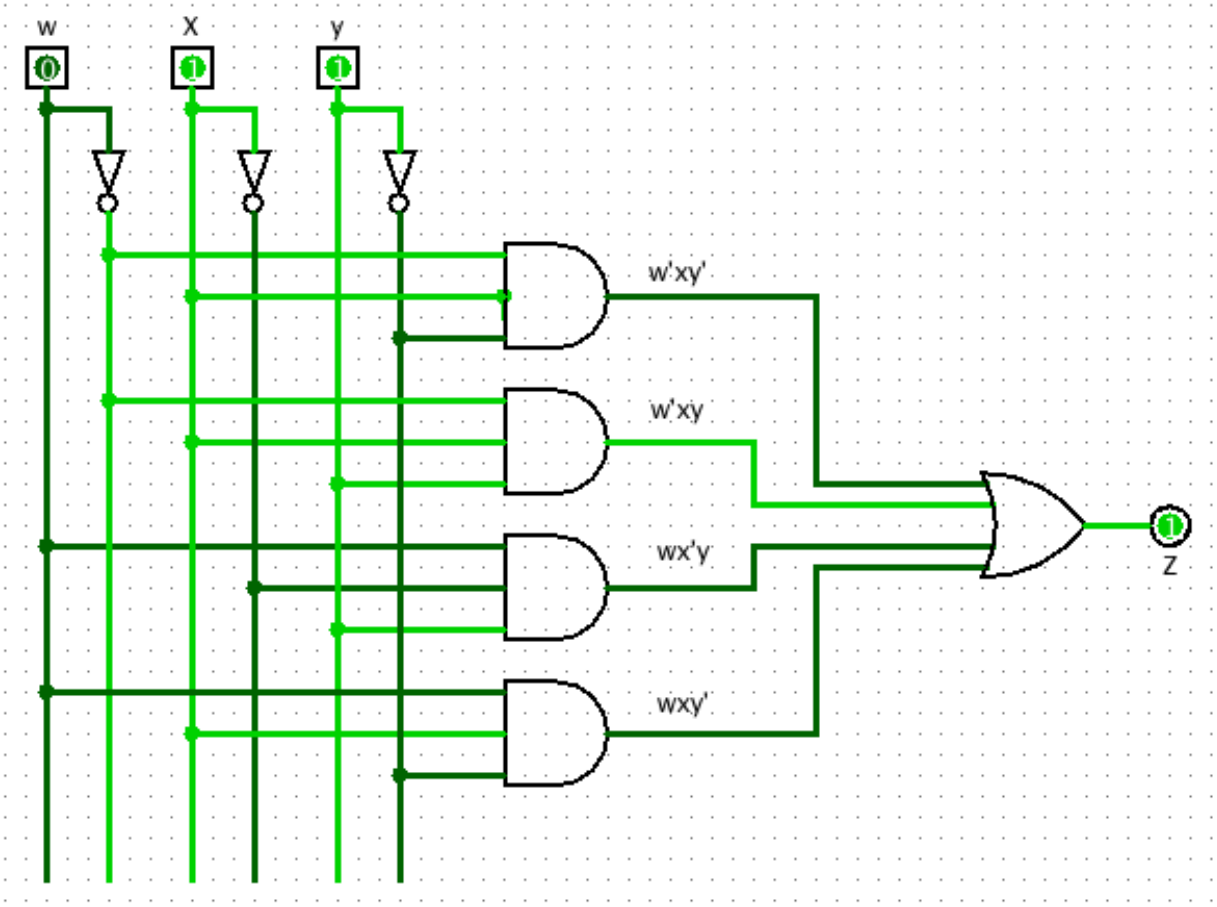
Illustrating Combinations



w'xy'

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

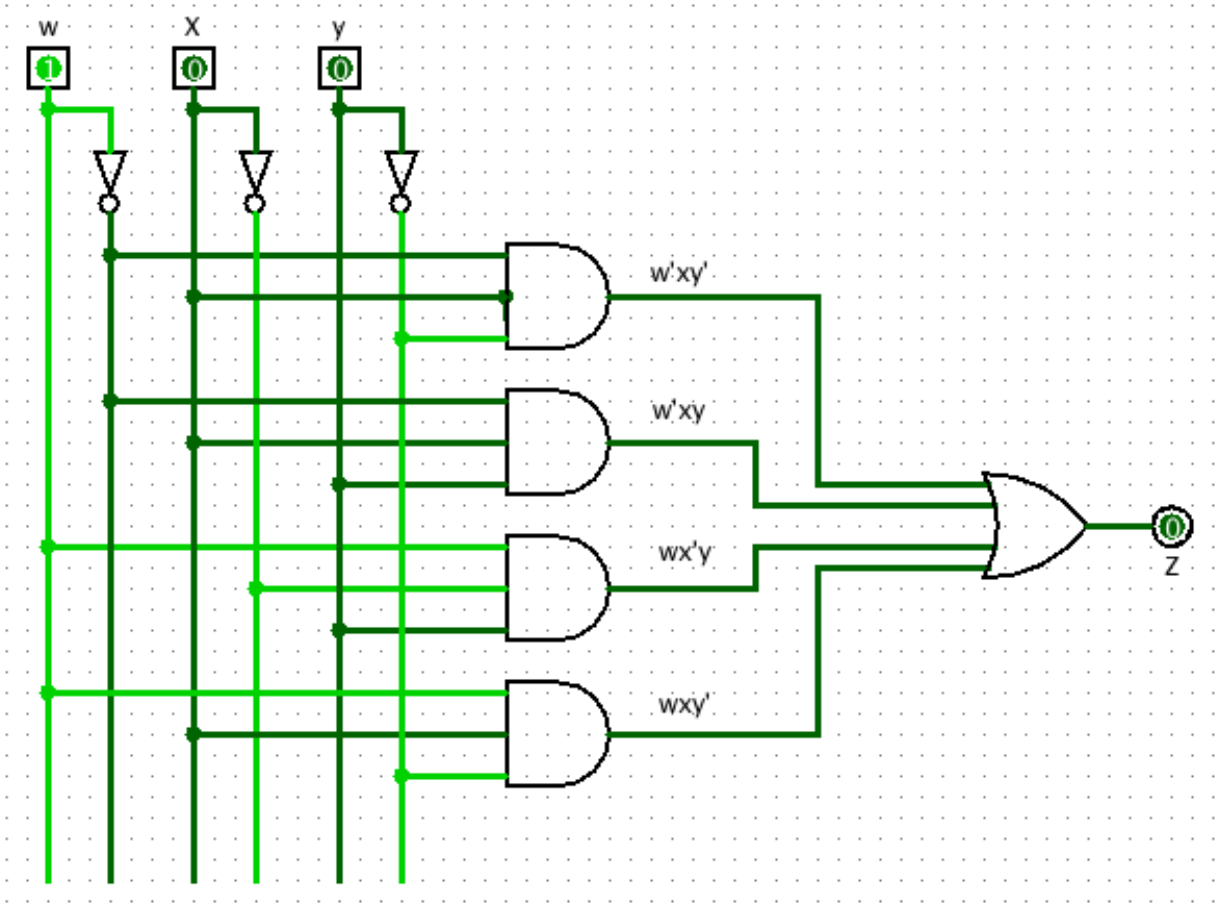
Illustrating Combinations



w'xy

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

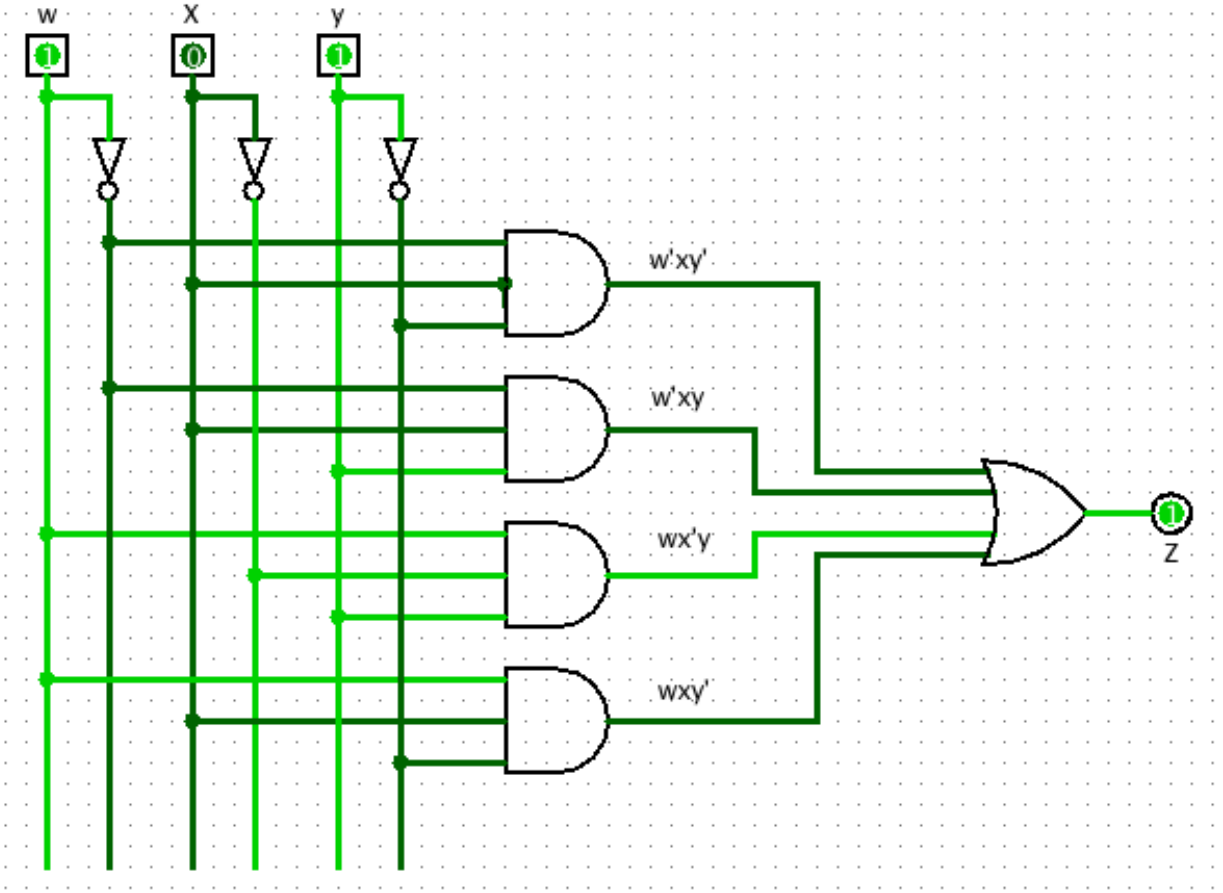
Illustrating Combinations



wx'y'

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

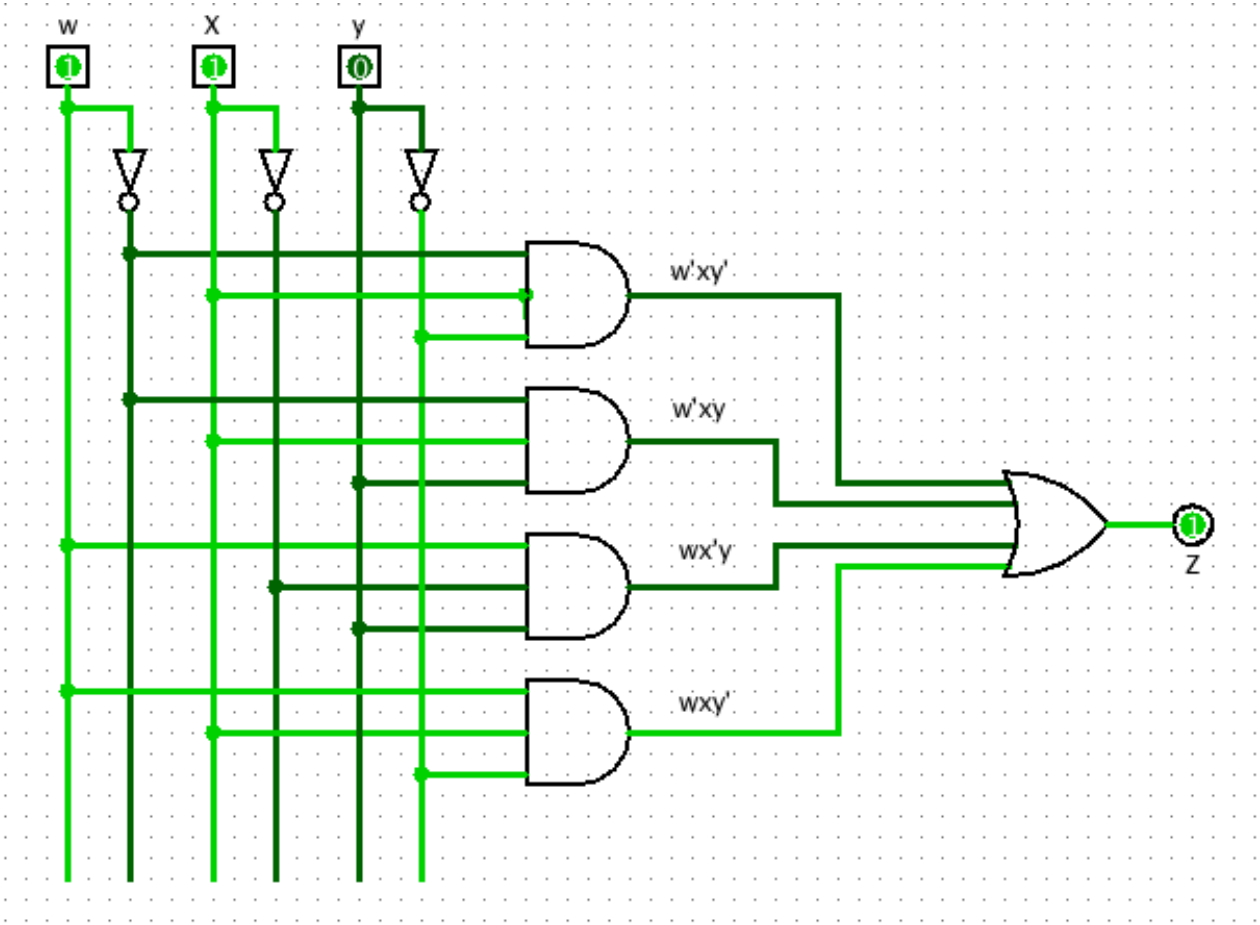
Illustrating Combinations



wx'y

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

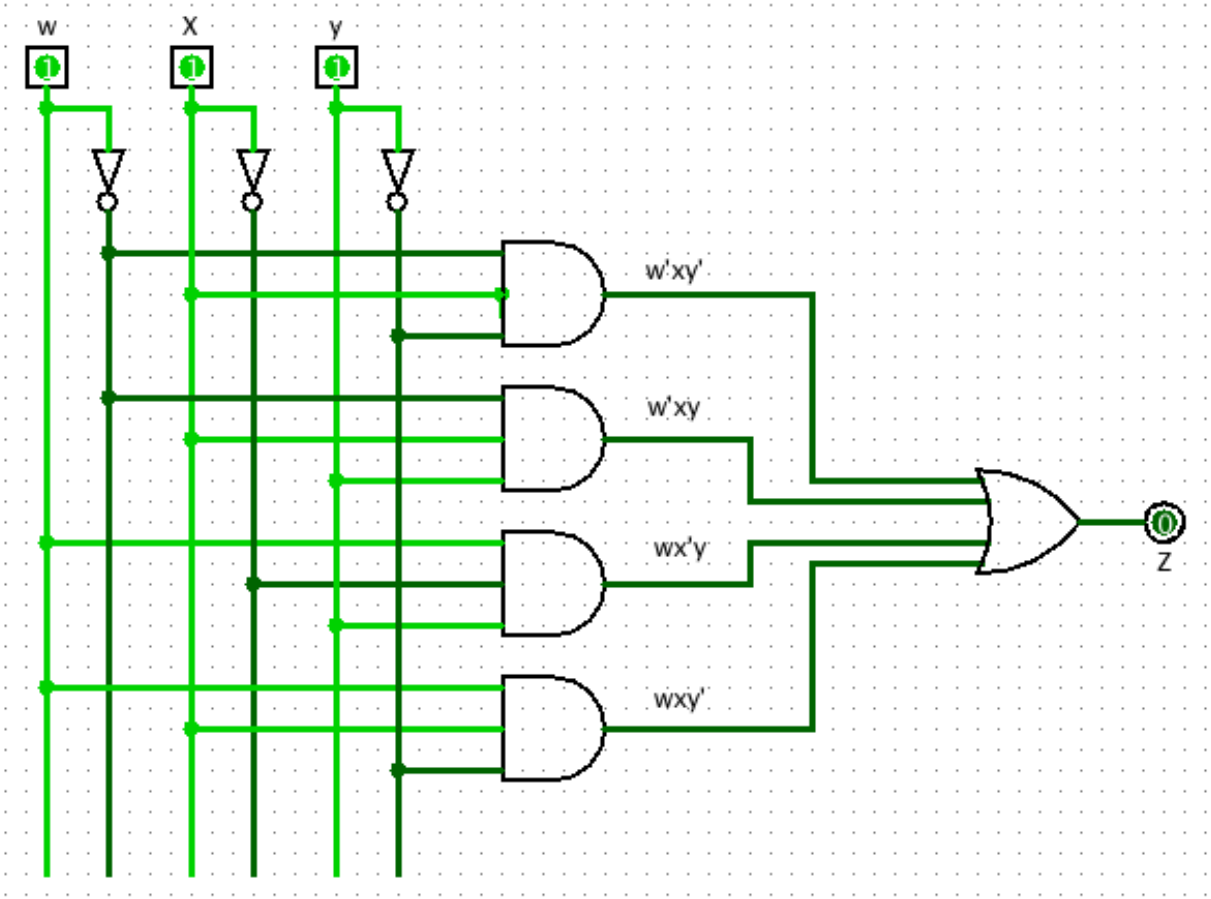
Illustrating Combinations



wxy'

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Illustrating Combinations



wxy

w	x	y	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A Universal Repertoire of One Gate Type

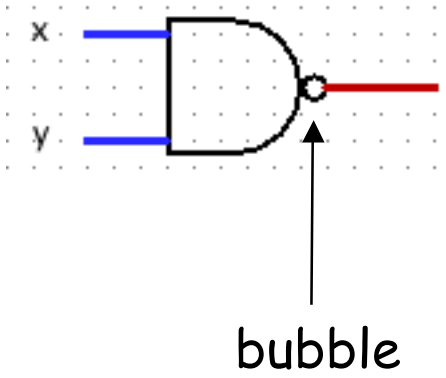
- nand = "not and"

nand

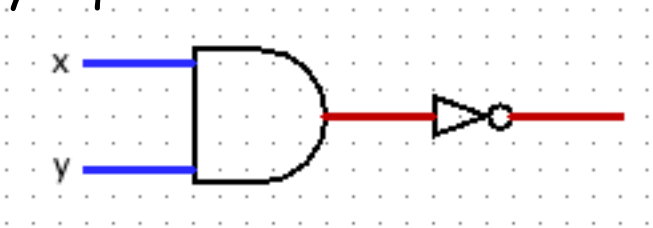
form 1 table:

x	y	nand(x, y)
0	0	1
0	1	1
1	0	1
1	1	0

gate symbol



logically equivalent to:



Universality of Nand

- Consider any minterm realization, e.g.

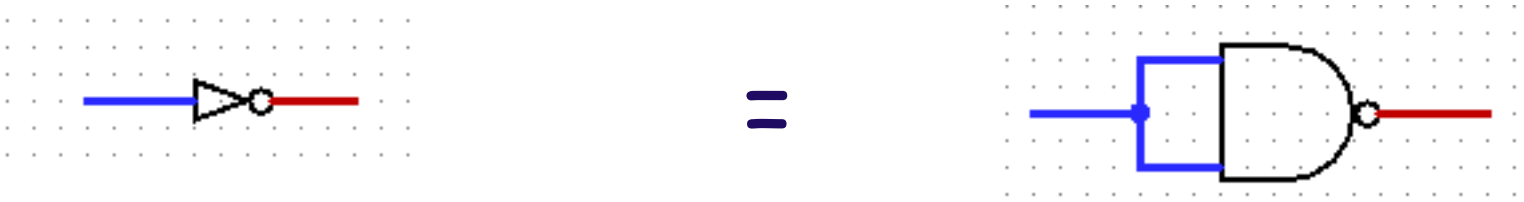
$$m_1 + m_2 + m_3$$

- To realize this with a final nand, use a version of deMorgan's law:

$$m_1 + m_2 + m_3 = (m_1' \, m_2' \, m_3')'$$
$$= \text{nand}(m_1', m_2', m_3')$$

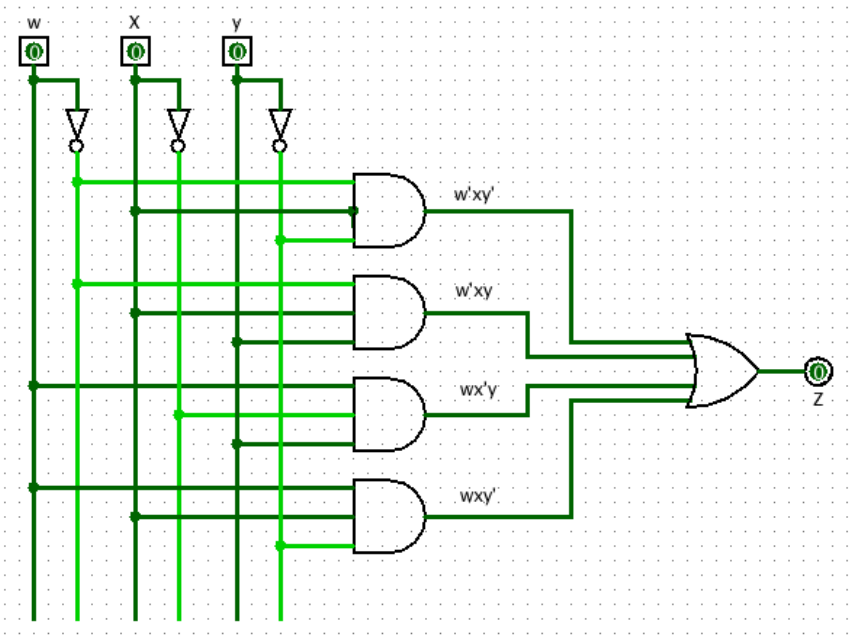
But each minterm m_i is an ***and*** of variables or their negations. So each m_i' is a ***nand*** of the same variables and their negations.

not from nand



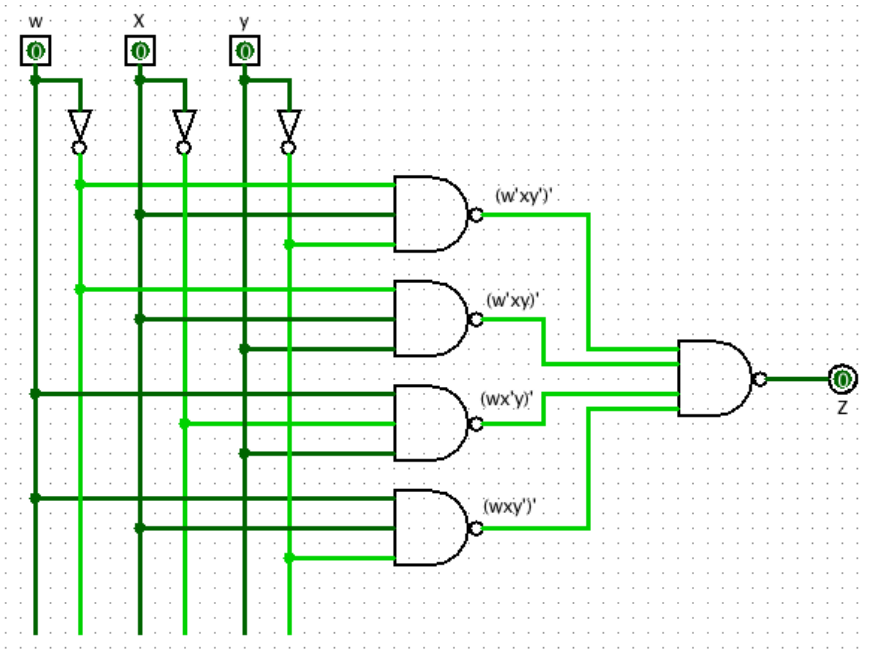
Nand Minterm Example

and/or



=

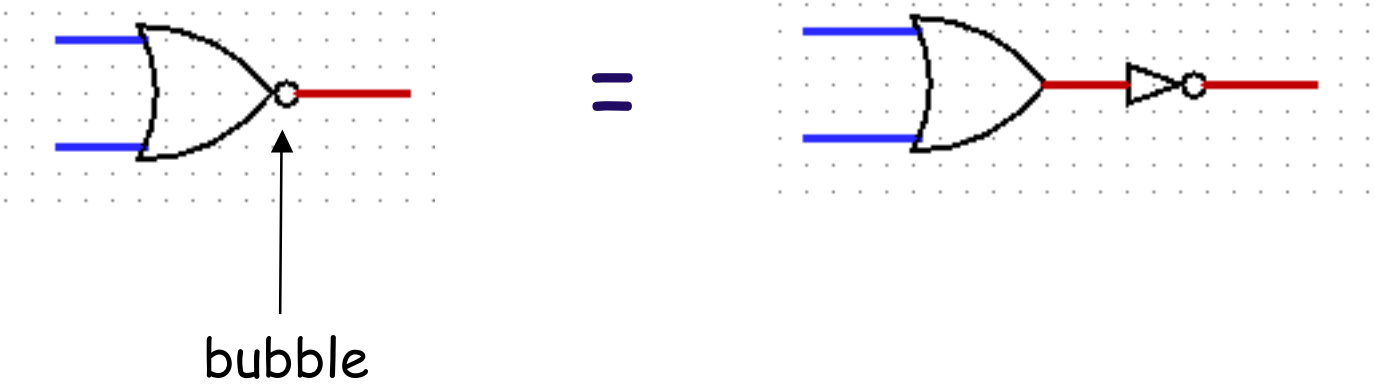
nand



Another universal set: {nor}

- *nor* is the negation of *or*
- Show that any minterm form can be re-expressed using only nor.

nor gate symbol



Number of Functions

- For n variables, how many minterms are there?
- How many distinct n-ary functions are there?

Building Blocks

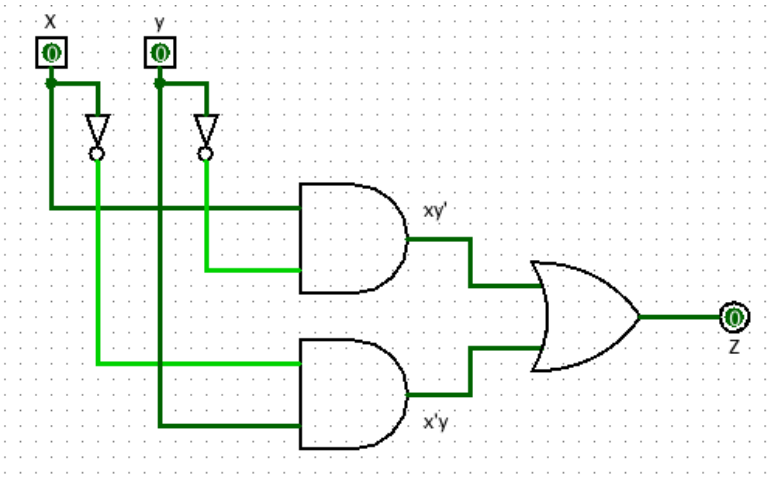
- For functions of large arity, it is sometime impractical to synthesize by previous methods: Tables may be too large.
- In such cases, it is common to use standard building blocks to construct circuits.

xor (exclusive or) building block

form 1 table:

x	y	xor(x, y)
0	0	0
0	1	1
1	0	1
1	1	0

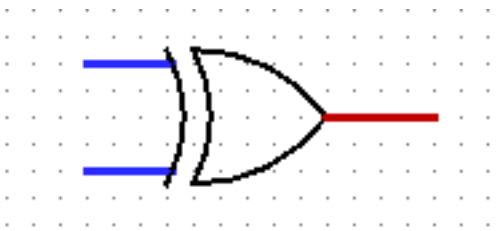
minterm realization

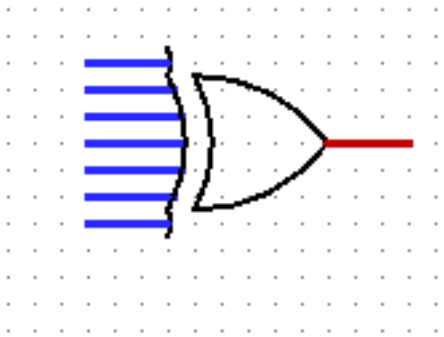


xor with >2 inputs

- Because *xor* is associative and commutative, the inputs can be regarded as a set with no particular order.

- xor symbols:





Full Adder (FA)

- The full adder is a building block sometimes used to construct adders of arbitrary length.

- Unlike the Half-Adder, an FA has:
  - 3 inputs: x, y, carryIn
  - 2 outputs: sum, carryOut

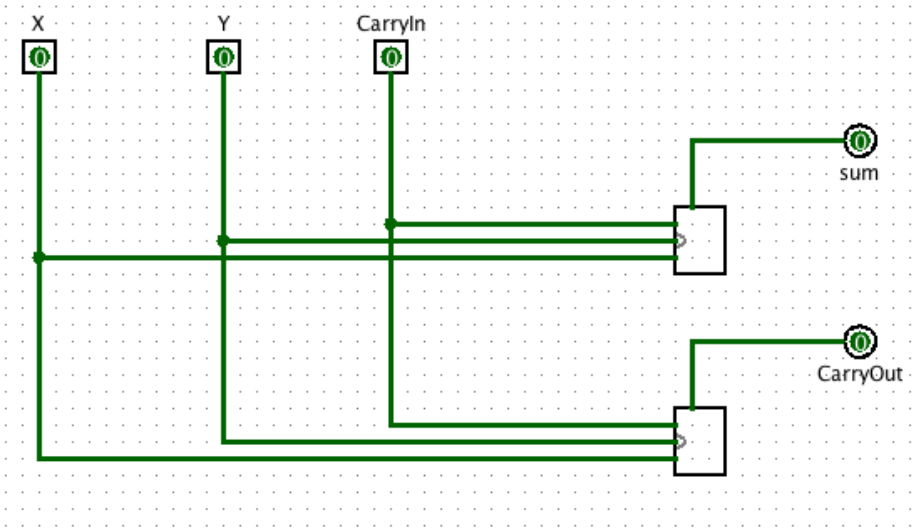
Full Adder (FA)

carryIn	x	y	carryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Each output
can be synthesized
as a separate function.

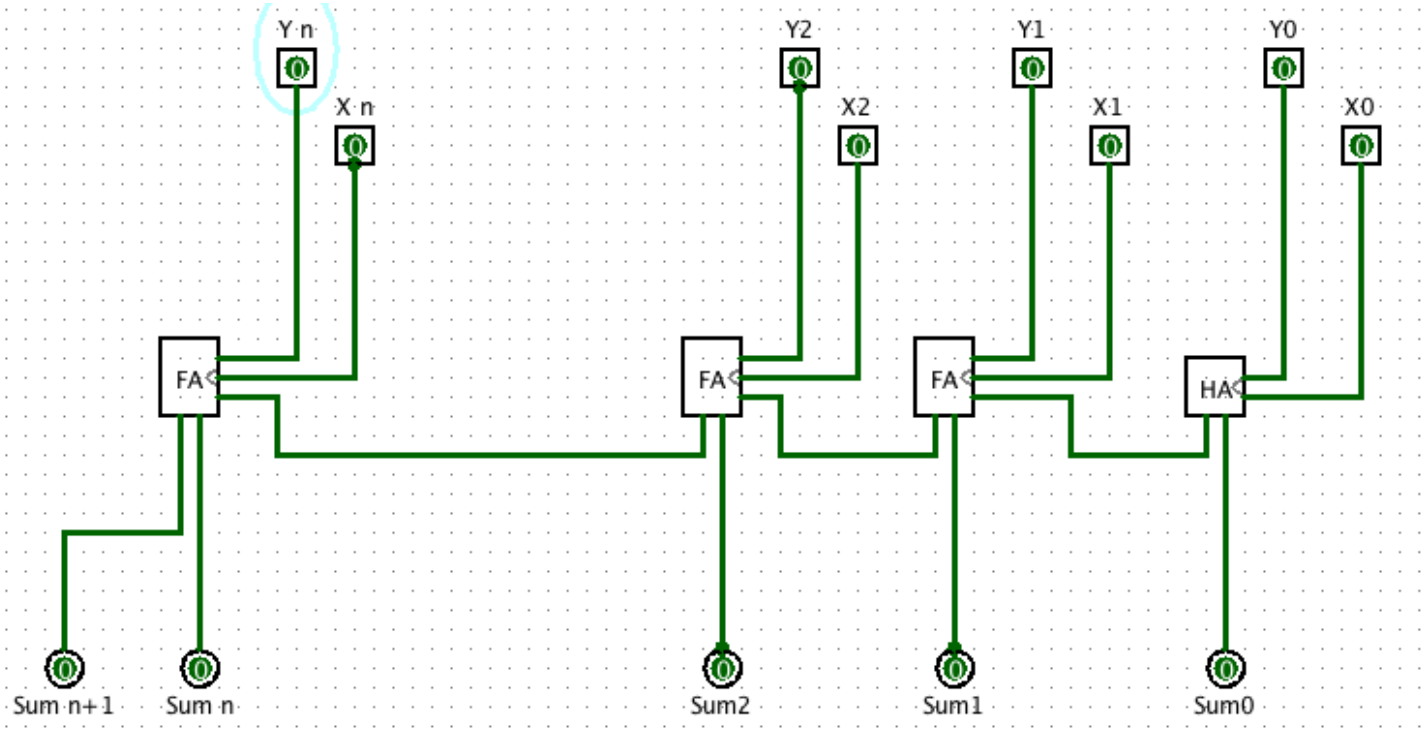
sum is a 3-input xor

CarryOut is a 2/3 majority



n-stage Ripple-Carry Adder

- This adder is easy to make, but slow for large n.



Gate-Depth

- The longest path in # of gates through the circuit determines the delay. This is called the "gate depth".

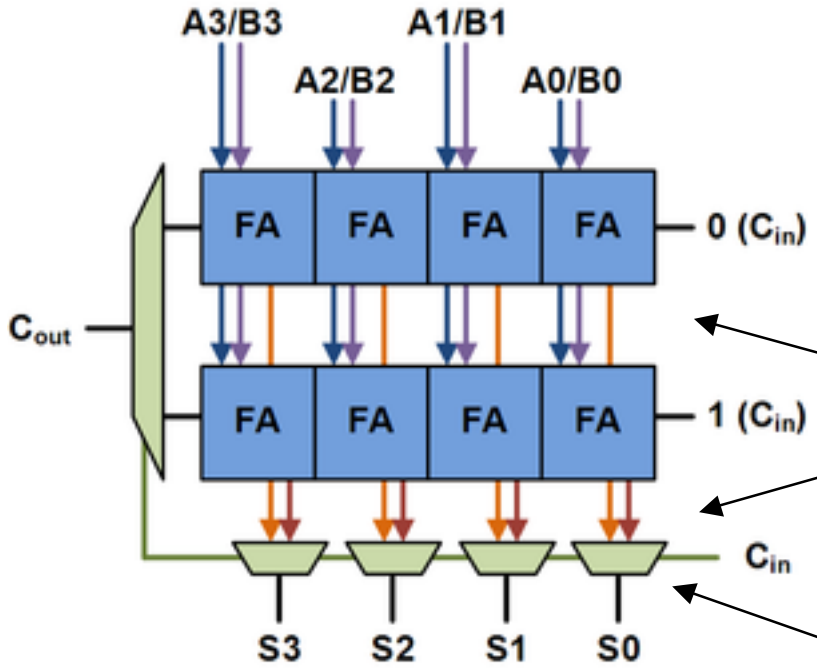
- An n-bit ripple carry adder has gate depth $O(n)$.

Faster Adders

- As an example of a faster adder, the carry-select adder has a gate depth of O(sqrt(n)).

- It works by computing outputs for ***both*** carryIn = 0 and carryIn = 1, then selecting the appropriate result when the carry is known.

Carry-Select Adder Building Block



Two contingent sums produced

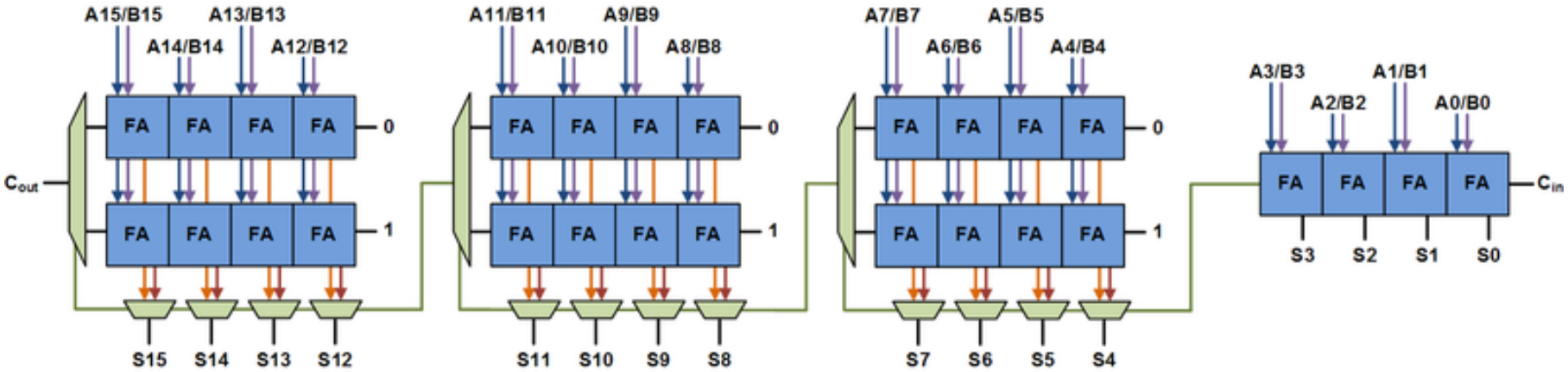
Selectors

http://en.wikipedia.org/wiki/Carry-select_adder

Carry-Select Adder Construction

carry-select blocks

ripple-carry starter



In general, sqrt(n) blocks of size sqrt(n) can be used.

Note that this is a Divide & Conquer approach.

http://en.wikipedia.org/wiki/Carry-select_adder

Even Faster Addition

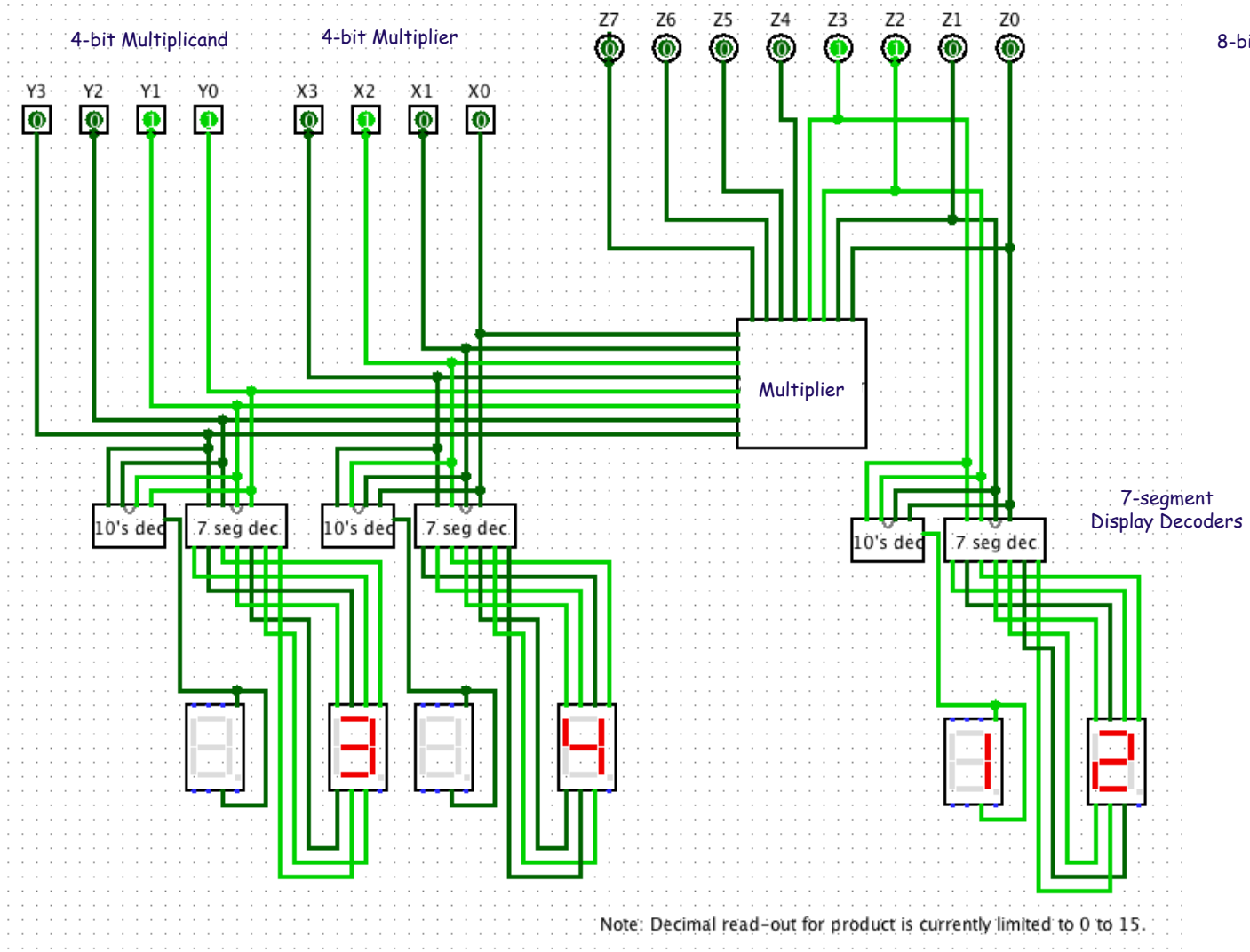
- A lower-bound for n-bit addition is $\Omega(\log n)$. (Ω, rather than O, is used for lower bounds).
- The Kogge-Stone adder achieves this bound.
- This is an example of a "digital" (radix-based) approach.

http://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder

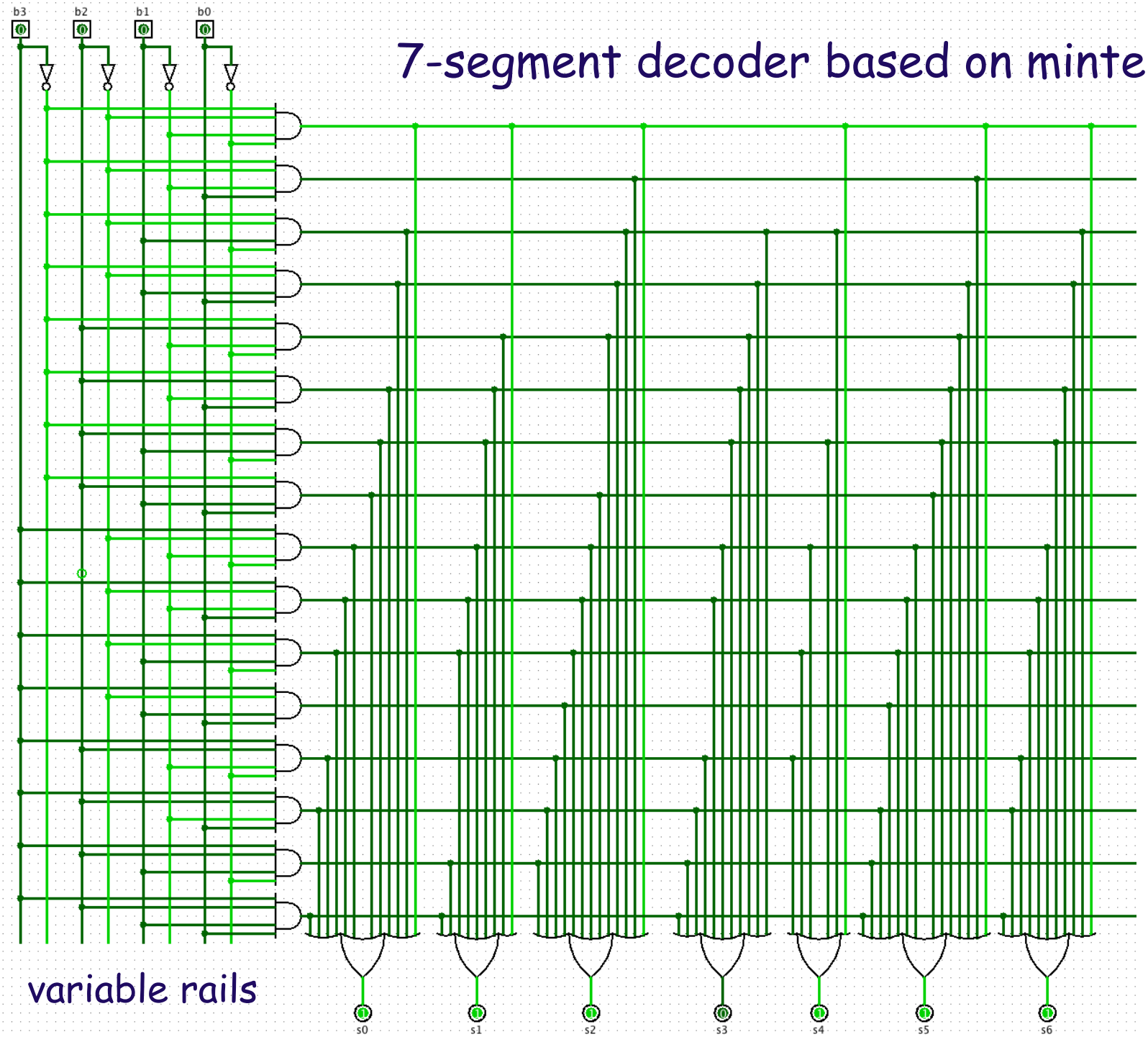
Multiplier Problem

- In the current assignment, you are asked to construct a multiplier.
- Consider using a "digital" method, wherein the multiplier's bits are used to decide whether or not to include the multiplicand at various points in the product computation.
- Essentially use the algorithm learned in school, but in binary.
- One approach is to use a "controlled" version of the ripple-carry adder as a building block.

A Partial Multiplier Tester



7-segment decoder based on minterms



variable rails

minterm rails

Programmable Logic Arrays (PLAs)

Elements with Memory

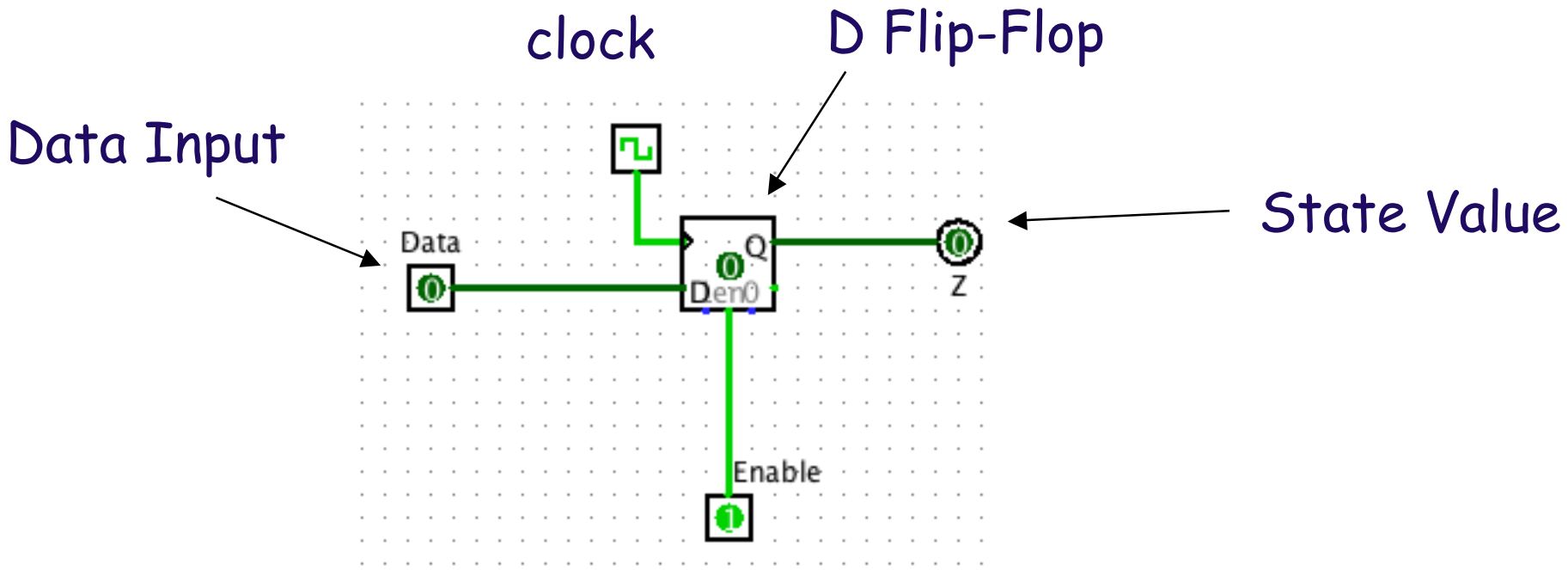
- So far we have discussed "combinational" elements. The output, after all signals have fully propagated, is a function of the input, without regard to history.

- More capabilities are available through elements with memory. These allow computations to be **sequential**, rather than just combinational.

Flip-Flop: A basic memory unit

- The flip-flop remembers the value (o or 1) it was last assigned.
- Assignment normally is done at the tick of a ***clock***.
- Circuits constructed this way are called **clocked sequential circuits**, or **synchronous circuits**.

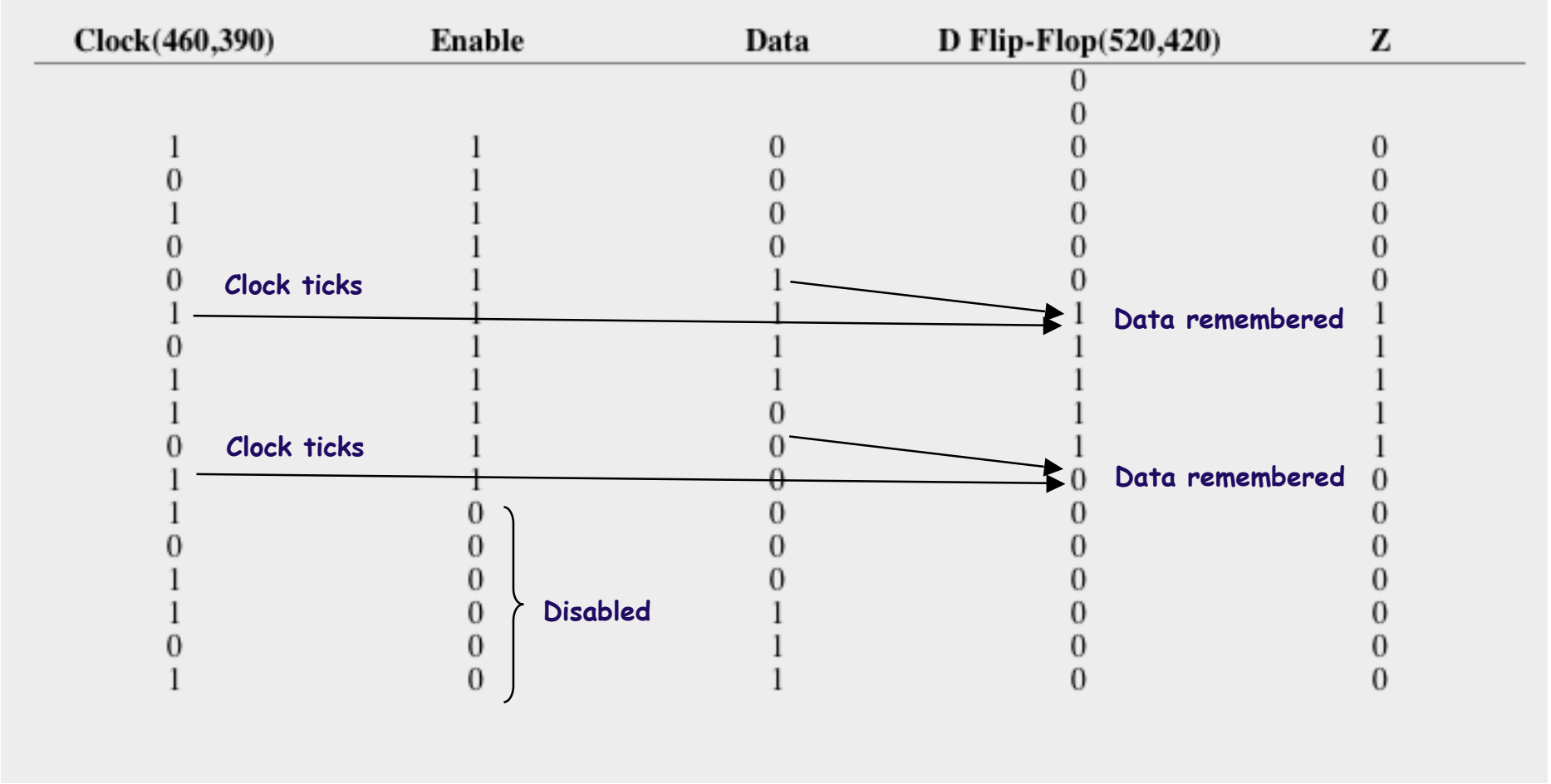
D Flip-Flop with Clock



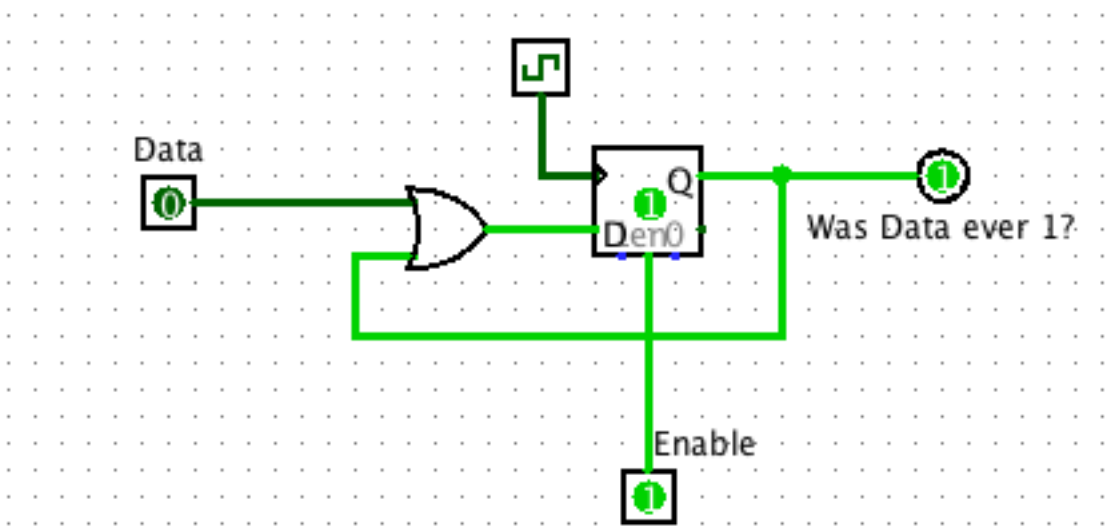
Enable Input

State can change only if enable = 1

D Flip-Flop Behavior



Sequential Circuit remembers whether Data was *ever* 1.



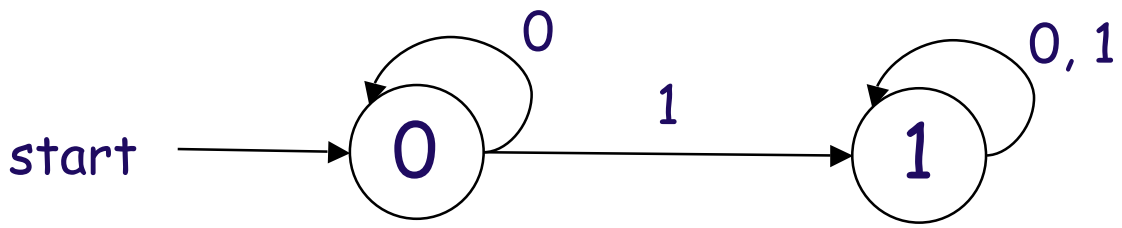
Clock(460,390)	D Flip-Flop(520,420)	Data	Enable	Was Data ever 1?
1	0	0	1	0
0	0	0	1	0
1	0	0	1	0
1	0	1	1	0
0	0	1	1	0
1	1	1	1	1
0	1	1	1	1
0	1	0	1	1
1	1	0	1	1
0	1	0	1	1
1	1	0	1	1
0	1	0	1	1

Clock ticks

Data remembered forever

State Diagram

Input = Value of Data at Tick



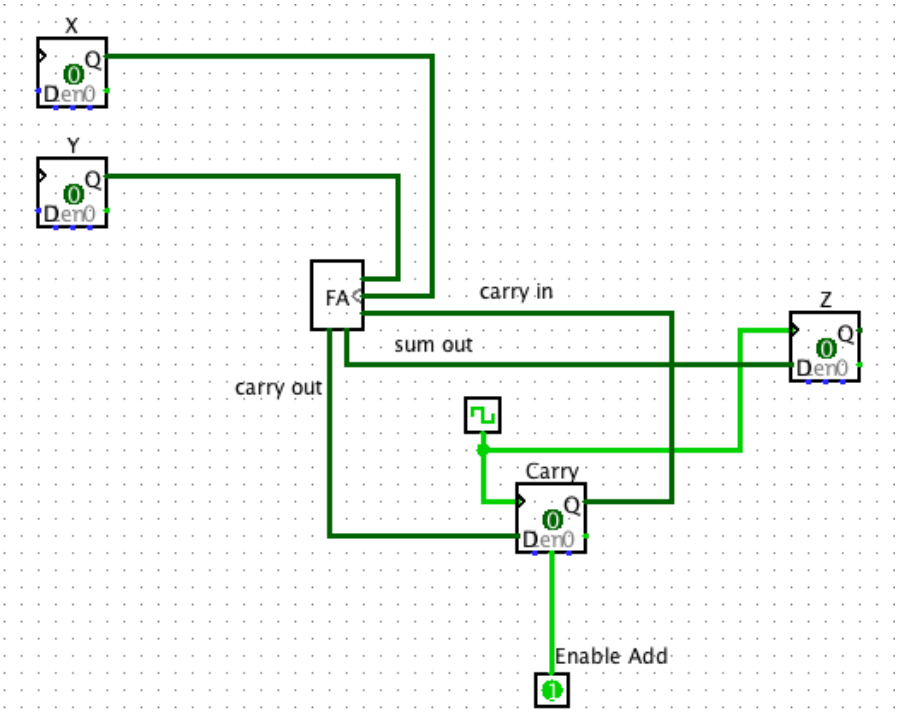
State = Value of Q

Sequential Adder

- Using a D Flip-Flop, the circuit can remember the carry from one tick to the next.
- Use two D Flip-Flops for input, one for output, one for carry.

Sequential Adder

adds X and Y least-significant bit first to get Z

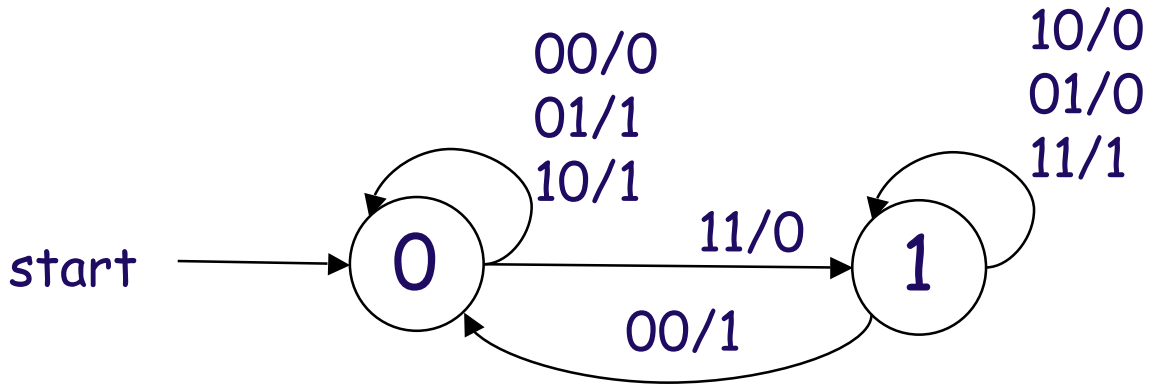


00111000110 X right to left
00111011100 Y right to left
01110100010 Z right to left
01111011100 Carry right to left

Clock(460,390)	Carry	X	Y	Z
0	0	0	0	0
1	0	0	0	0
0	0	0	0	0
0	0	1	0	0
1	0	1	0	1
1	0	0	0	1
0	0	0	0	1
0	0	1	0	1
0	0	1	1	1
1	1	1	1	0
0	1	1	1	0
0	1	0	1	0
1	1	0	1	0
0	1	0	1	0
1	1	0	1	0
1	1	0	0	0
0	1	0	0	0
1	0	0	0	1
0	0	0	0	1
0	0	1	0	1
0	0	1	1	1
1	1	1	1	0
0	1	1	1	0
1	1	1	1	1
0	1	1	1	1
1	1	1	1	1
0	1	1	1	1
0	1	0	1	1
0	1	0	0	1
1	0	0	0	1
0	0	0	0	1
1	0	0	0	0
0	0	0	0	0

State Diagram for Adder

Input/Output = XY/Z



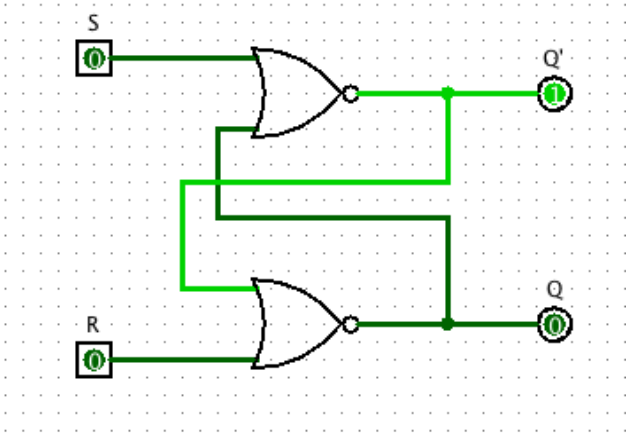
State = Value of Carry Stored

Flip-Flop Construction

- Flip-Flops can be constructed from basic gates.
- Broad-categories, of increasing sophistication, are:
  - Latch
  - Gated Latch, or Level-Triggered Flip-Flop
  - Edge-triggered Flip-Flop

Latches

- A latch is the basic memory structure for a flip-flop.
- Here is one possible latch, called set-reset (SR), constructed from **nor** gates:

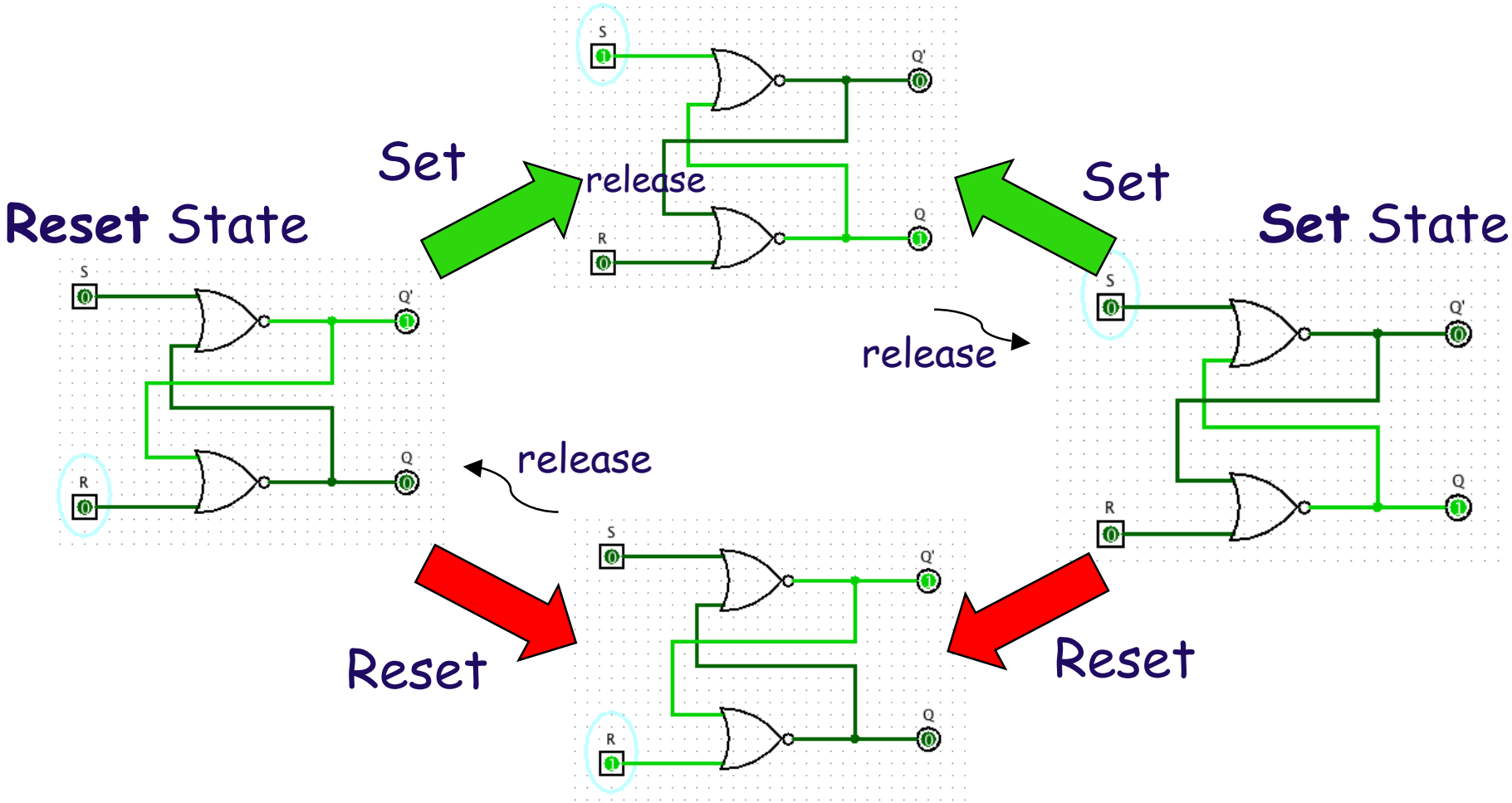


Shown is the **reset** state:
Q = 0, Q' = 1.

Latch Operation

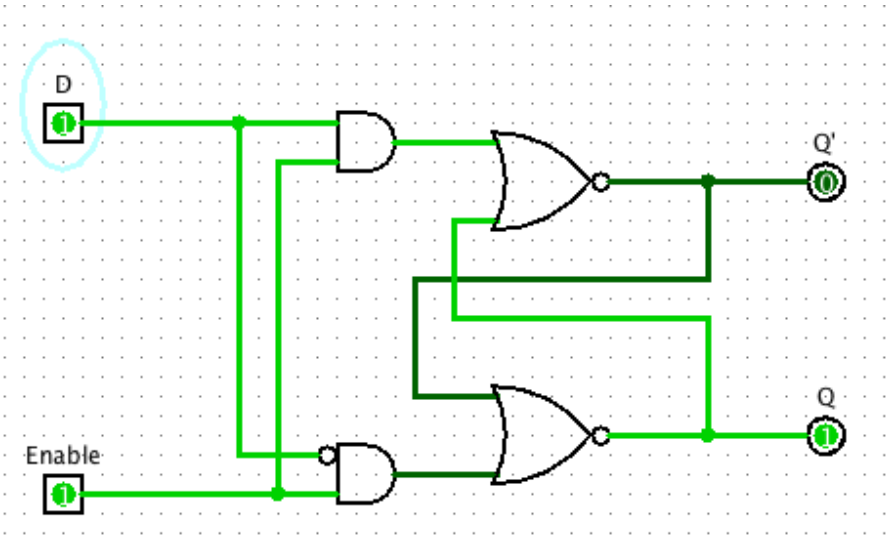
- Only one of S, R should be 1 at any given time.
- When S is raised, the latch goes to the **set** state (Q = 1, Q' = 0).
- When S is lowered, the set state is maintained.
- When R is raised, the latch goes to the **reset** state (Q = 0, Q' = 1).
- When R is lowered, the set state is maintained.

Latch Operation



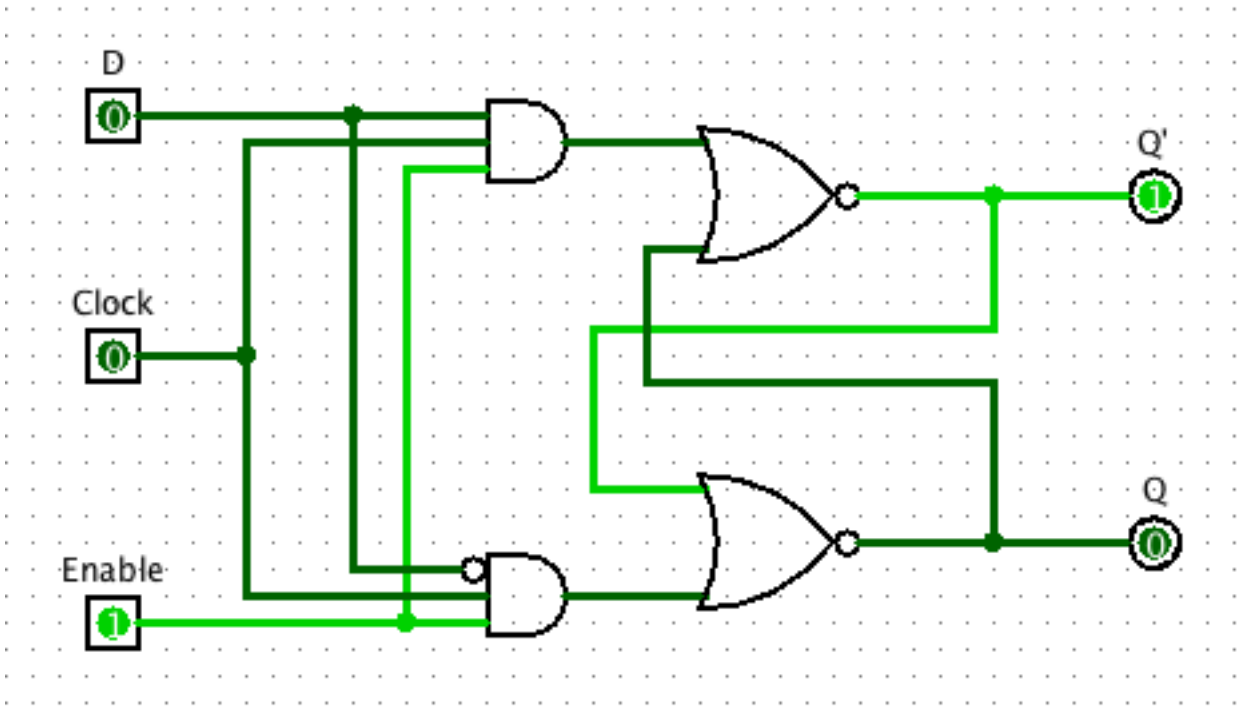
Gated Latch

- This enables or disables the latch set and reset.
- Enable line is also called "Strobe".
- D is for "Data"

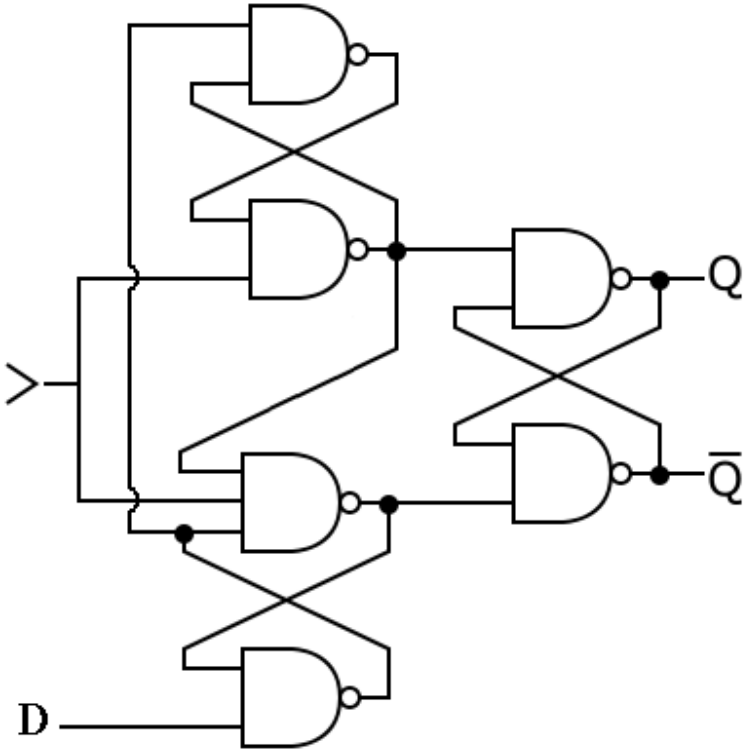


Level-Triggered Flip-Flop

- Adds clock input to gated latch
- Enable also called "Strobe"



Edge-Triggered Flip-Flop

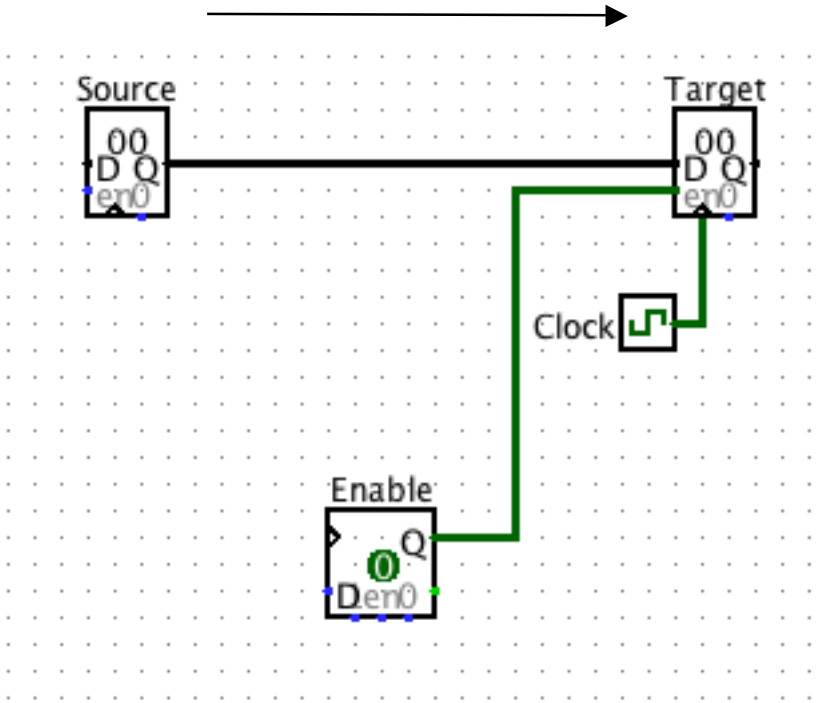


http://en.wikipedia.org/wiki/Flip-flop_(electronics)

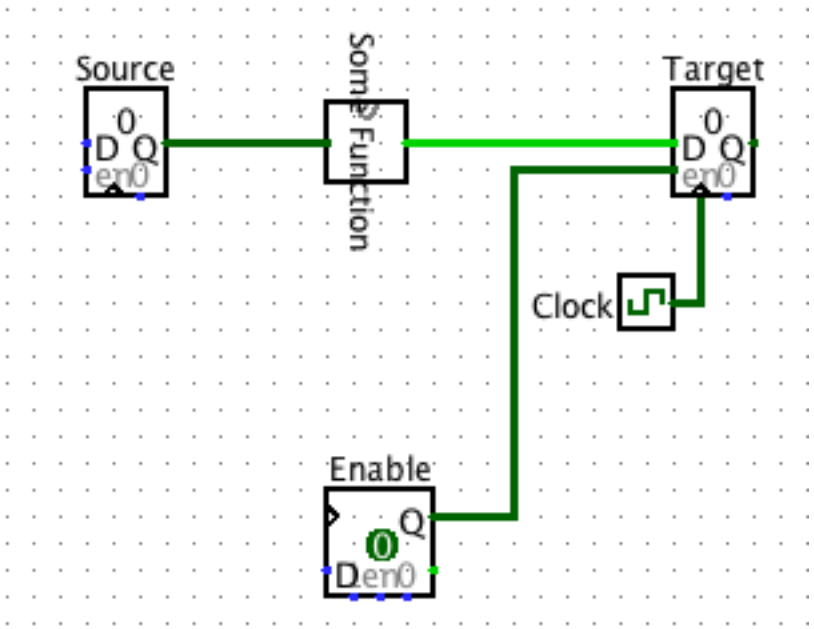
Registers

- A flip-flip is a 1-bit register.
- Flip-flops can be combined in parallel to produce multiple-bit registers.
- Contents of one register can be transferred to another on clock tick

Straight Register Transfer

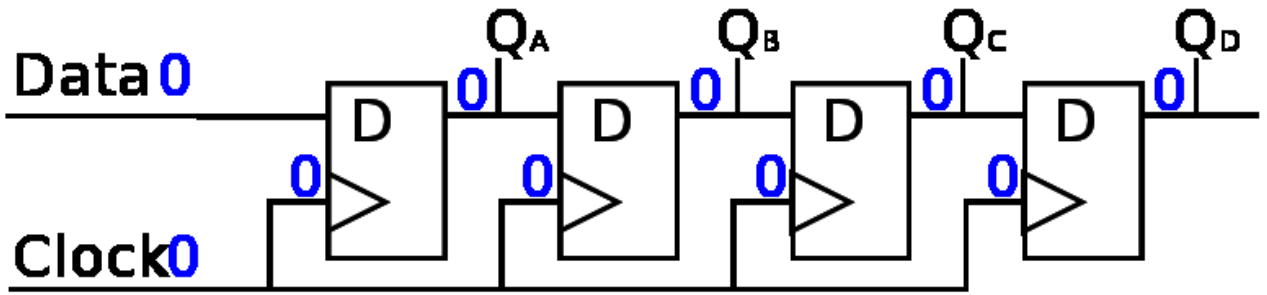


Register Transfer with Combinational Functions



Extra In-Register Functions

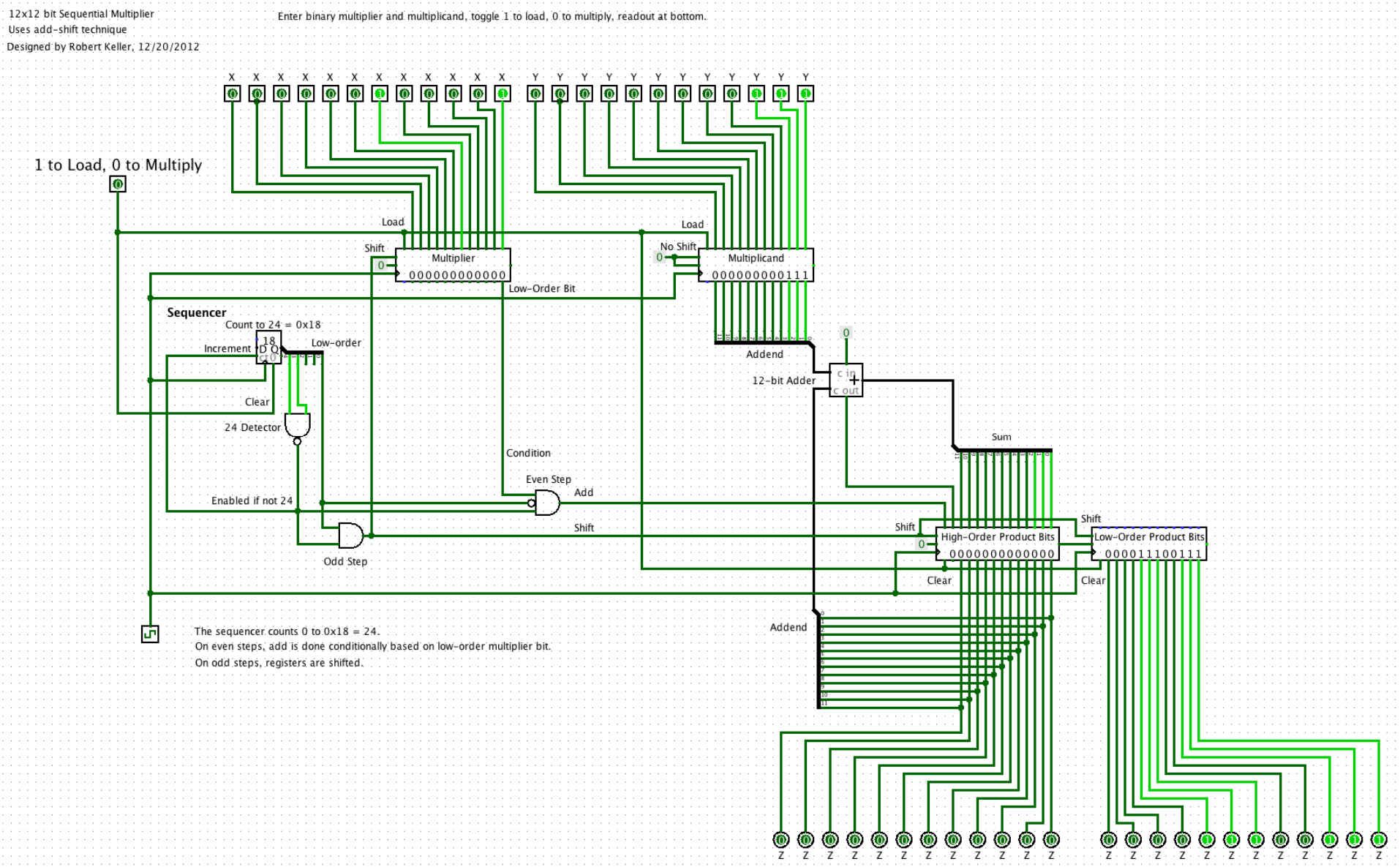
- Shift (Left or Right)
- Increment or Decrement
- Zero
- Complement



Shift Register

http://en.wikipedia.org/wiki/Flip-flop_(electronics)

Sequential Multiplier



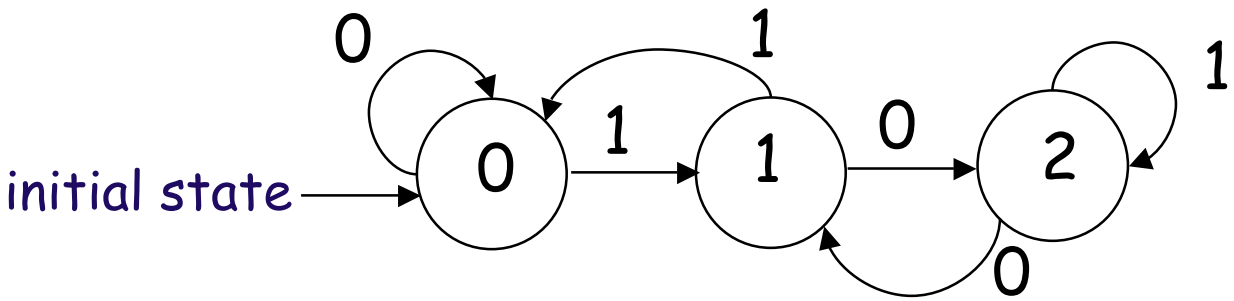
Implementing General Finite-State Machines

- Finite-State machines have many uses in computers, including registers and sequencers.
- Here we deal with the general problem of implementing an FSM starting with a state-transition diagram.

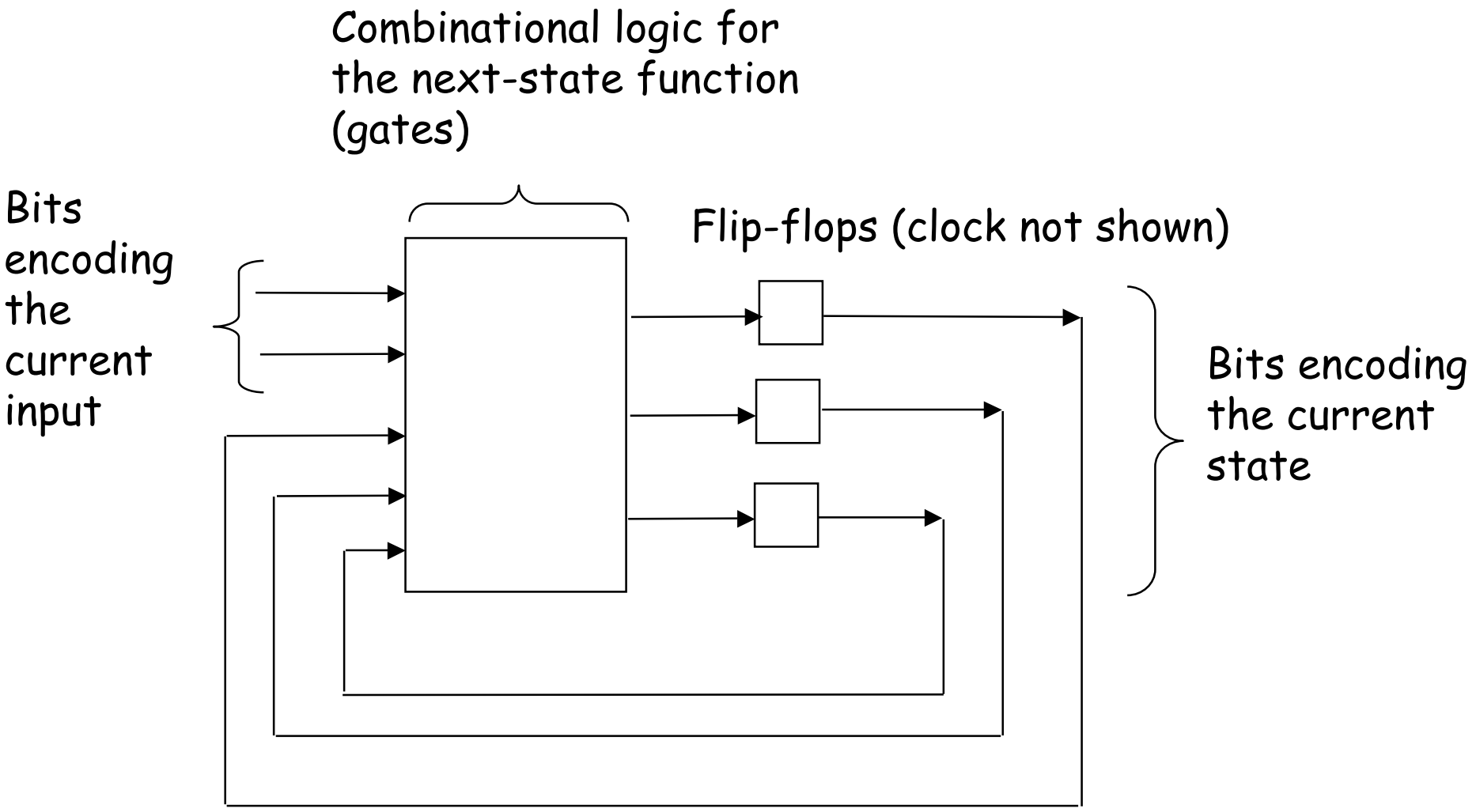
Memory Considerations

- The state of the machine has to be stored in some kind of memory, usually flip-flops.
- The flip-flop values represent an **encoding** of the current state.

State-Machine Example



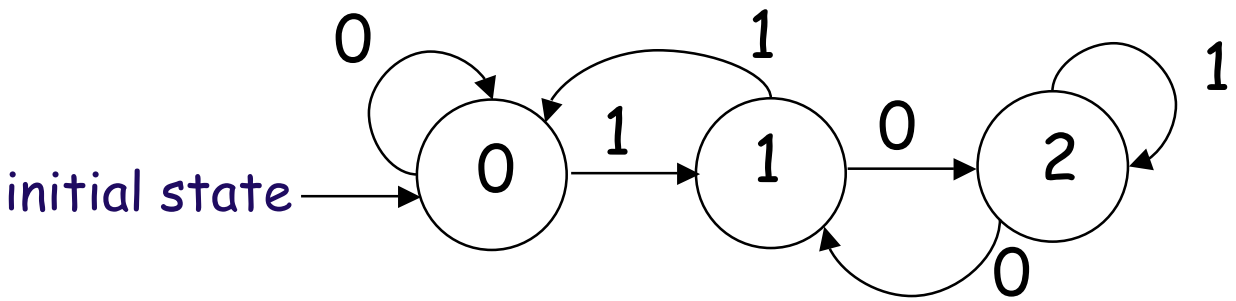
Target FSM Architecture



Input and State Encodings

- Encodings may be pre-specified, or at the discretion of the designer.

- Example: Suppose the input is pre-specified to be {0, 1}, while the state encodings are discretionary.



Encoding State Set

- State set is {0, 1, 2}.
- What are the possible ways to encode it in binary?
- What is the fewest number of bits?

Binary Encoding

- 0 → 00
- 1 → 01
- 2 → 10

(→ = "is encoded as")

Gray-Code Encoding

- 0 → 00
- 1 → 01
- 2 → 11

General Gray-Code Pattern: Reflection

1-bit

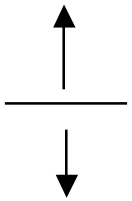
0
1

2-bit

0	0
0	1
1	1
1	0

3-bit

0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0



One-Hot Encoding

- 0 → 100
- 1 → 010
- 2 → 001

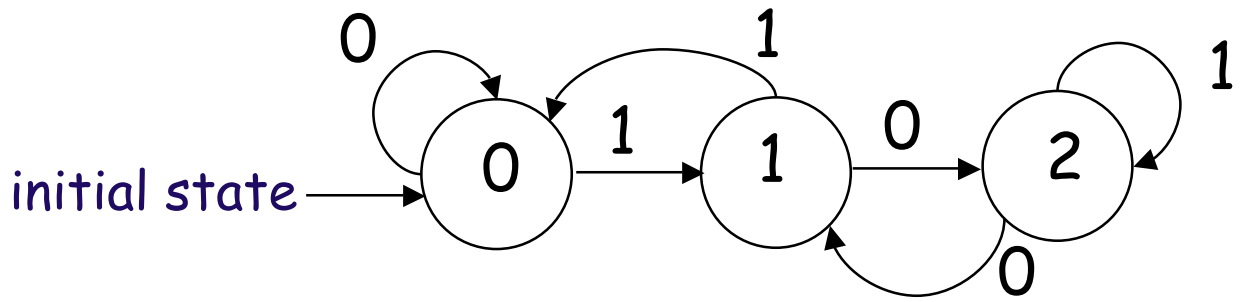
"Thermometer" Encoding

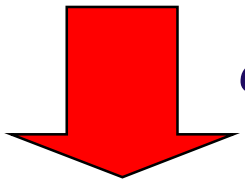
- 0 → 100
- 1 → 110
- 2 → 111

Simplest Logic

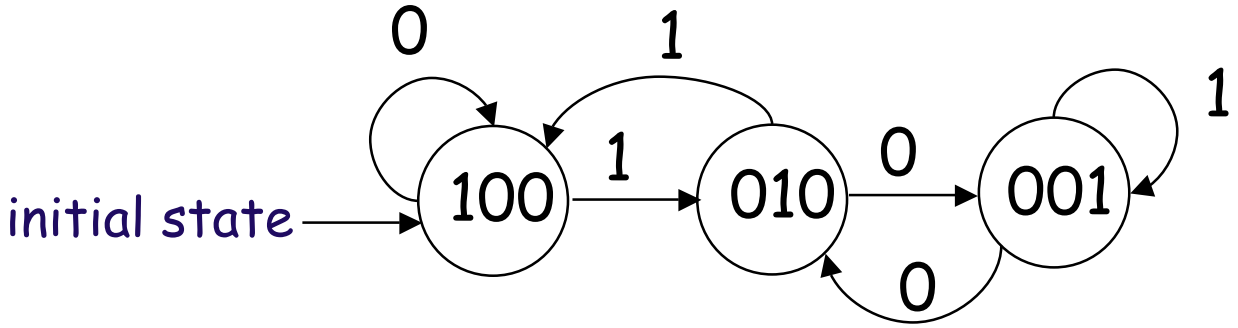
- One-Hot Encoding often yields simplest design.
- However, it is expensive if the number of states is large (n flip-flops for n states, vs. ceil(log(n)) possible).

One-Hot Version

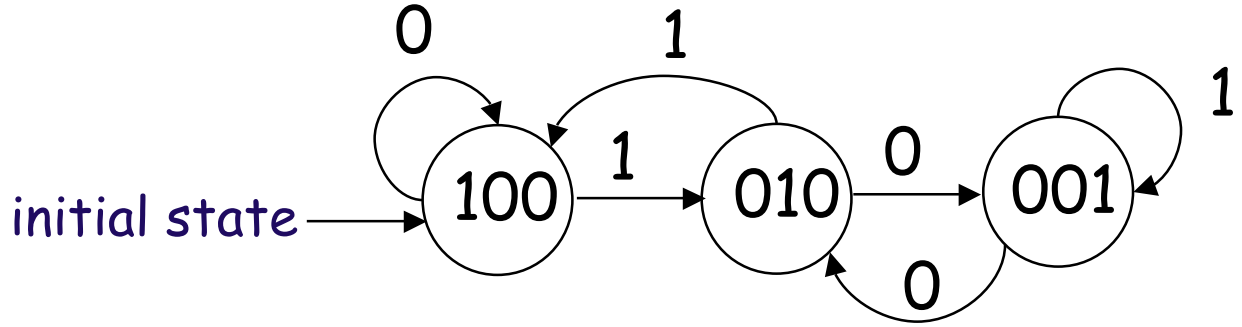




encode states

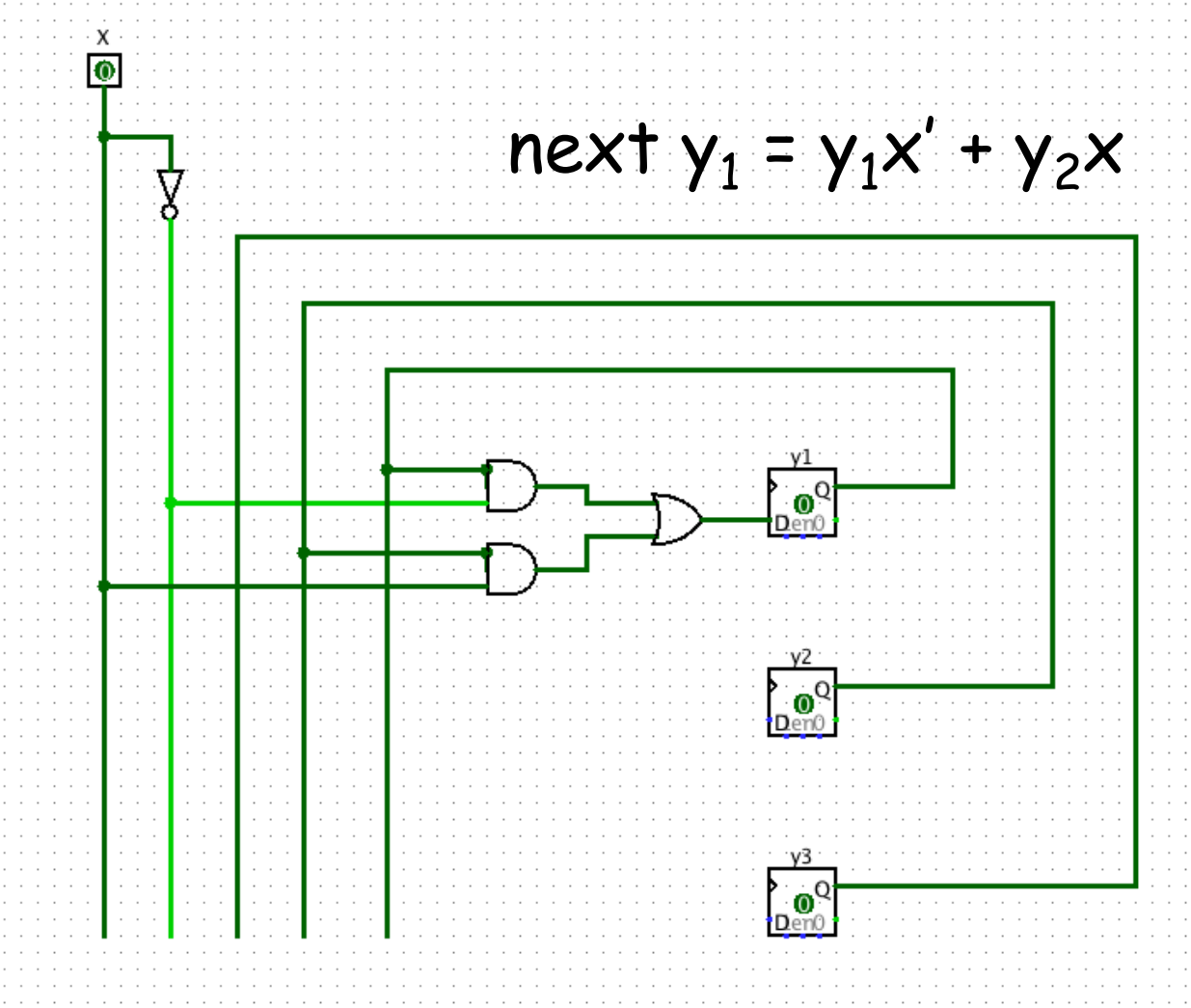


For One-Hot, logic functions can be **read off** from the state diagram

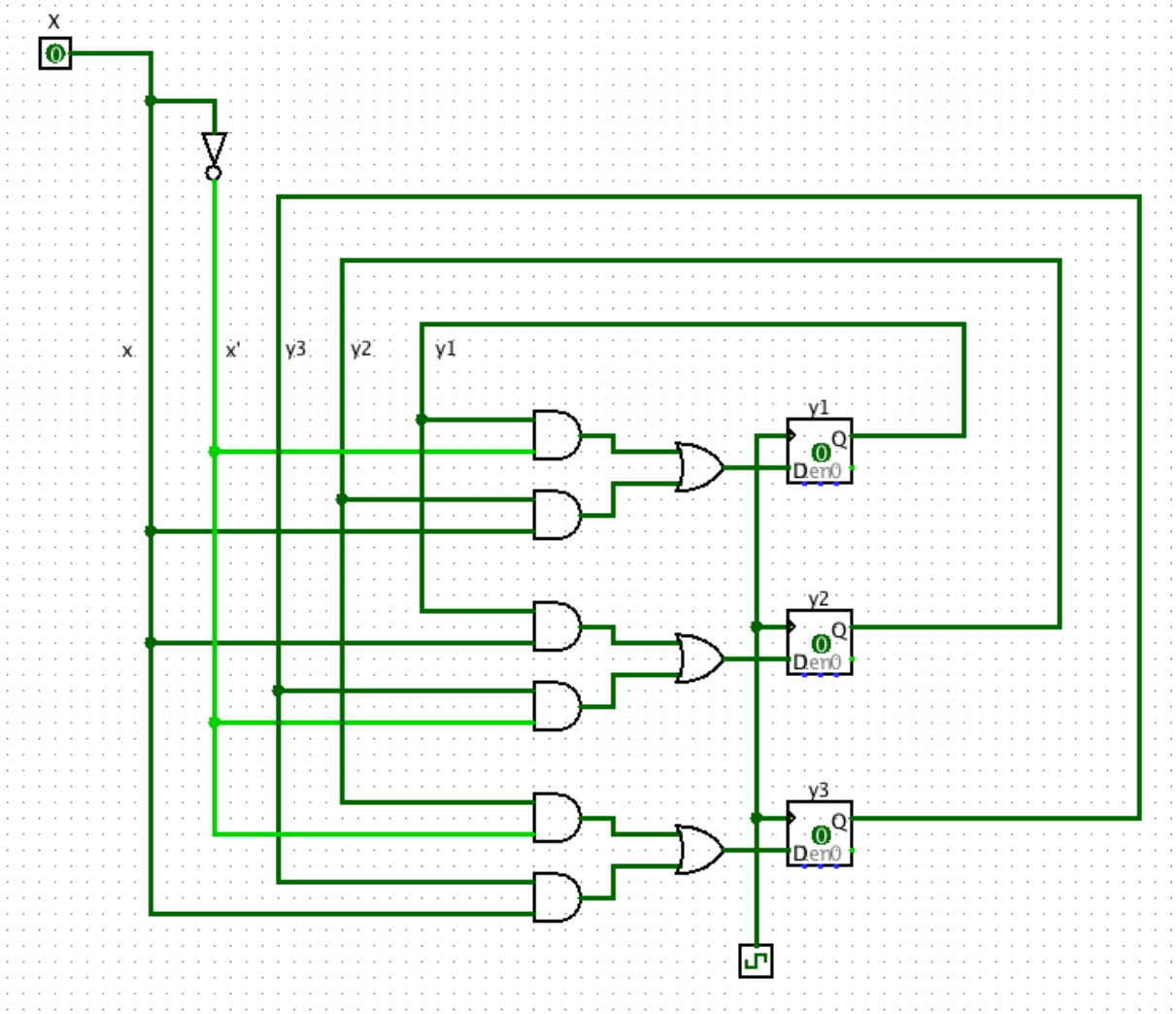


- Let the input variable be x.
- Let the state variables be $y_1y_2y_3$ (initially 100).
- Then in combinational logic:
  - next $y_1 = y_1x' + y_2x$
  - next $y_2 = y_1x + y_3x'$
  - next $y_3 = y_2x' + y_3x$
- Note that in a one-hot design, the negations of flip-flip values are not needed.

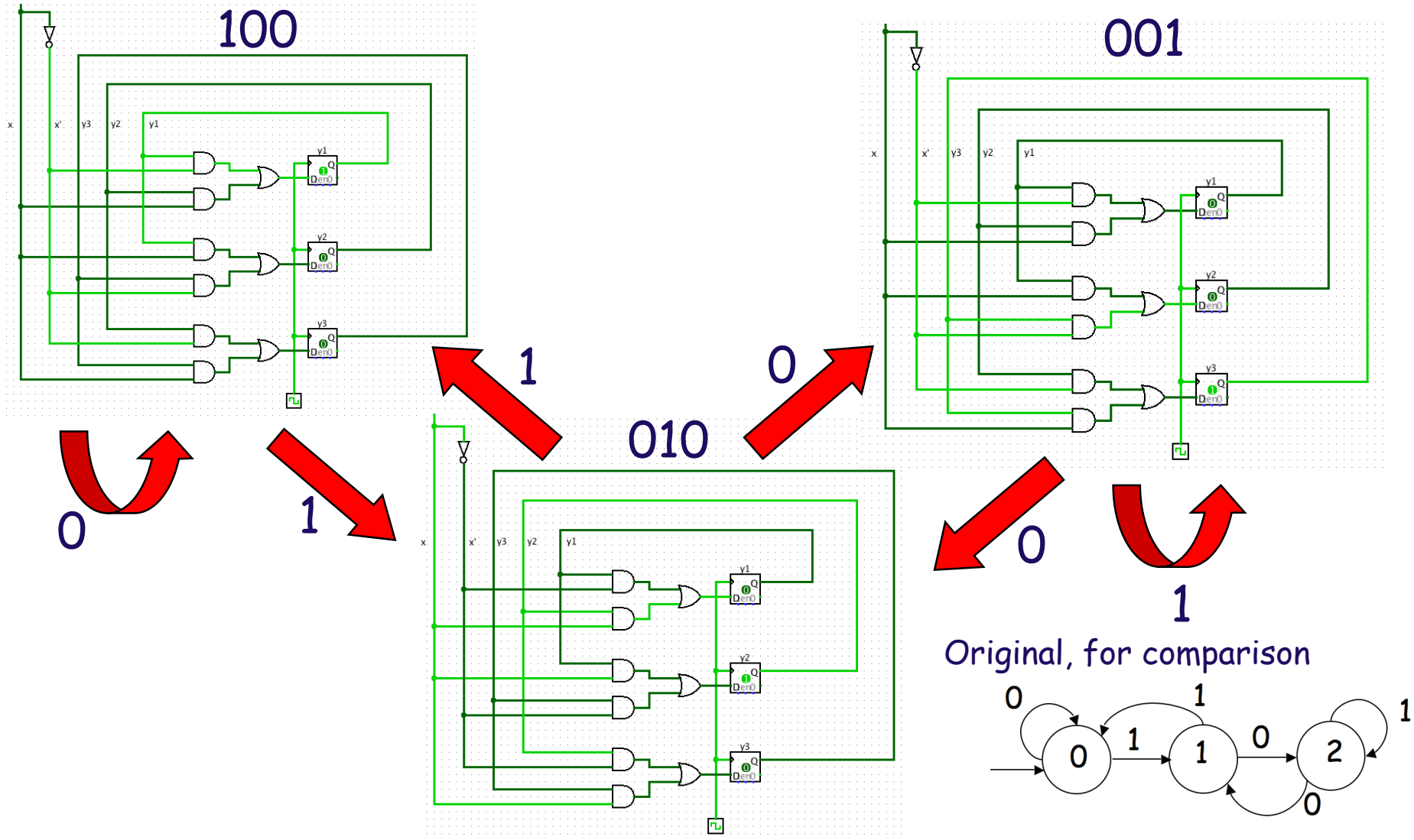
First Equation



Final Design



Check Operation

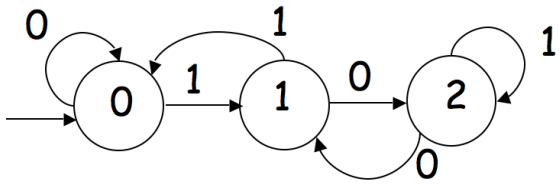


Using a Different State Encoding

- To demonstrate, let's use the Gray-code encoding:
  - 0 → 00
  - 1 → 01
  - 2 → 11

Transition Table

- Here is the state-transition function in tabular form:



Next State		Input	
		0	1
Current State	0	0	1
	1	2	0
	2	1	2

Edit the States

- Carefully replace each state with its corresponding encoding:
  0 → 00, 1 → 01, 2 → 11

Next State		Input	
		0	1
Current State	00	00	01
	01	11	00
	11	01	11

Split the Table into Separate Tables for Each Flip-Flop

Next State		Input 0	Input 1
Current State	00	00	01
	01	11	00
	11	01	11

Next y_1		Input 0	Input 1
Current State	00	0	0
	01	1	0
	11	0	1

Next y_2		Input 0	Input 1
Current State	00	0	1
	01	1	0
	11	1	1

Read Off Equations from Each Table

Next y_1		Input	
		0	1
Current State	00	0	0
	01	1	0
	11	0	1

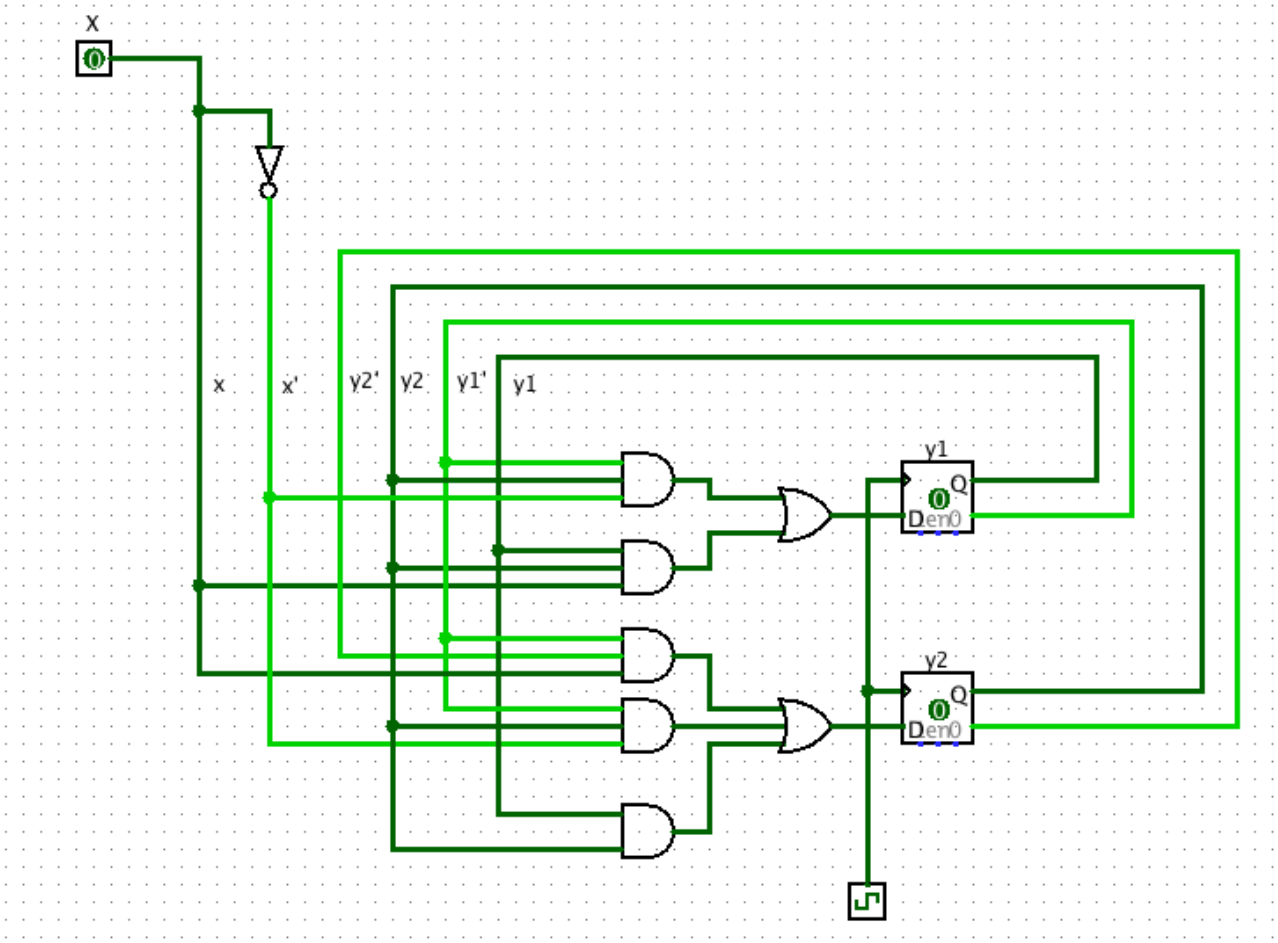
Next y_2		Input	
		0	1
Current State	00	0	1
	01	1	0
	11	1	1

$$\text{next } y_1 = y_1'y_2x' + y_1y_2x$$

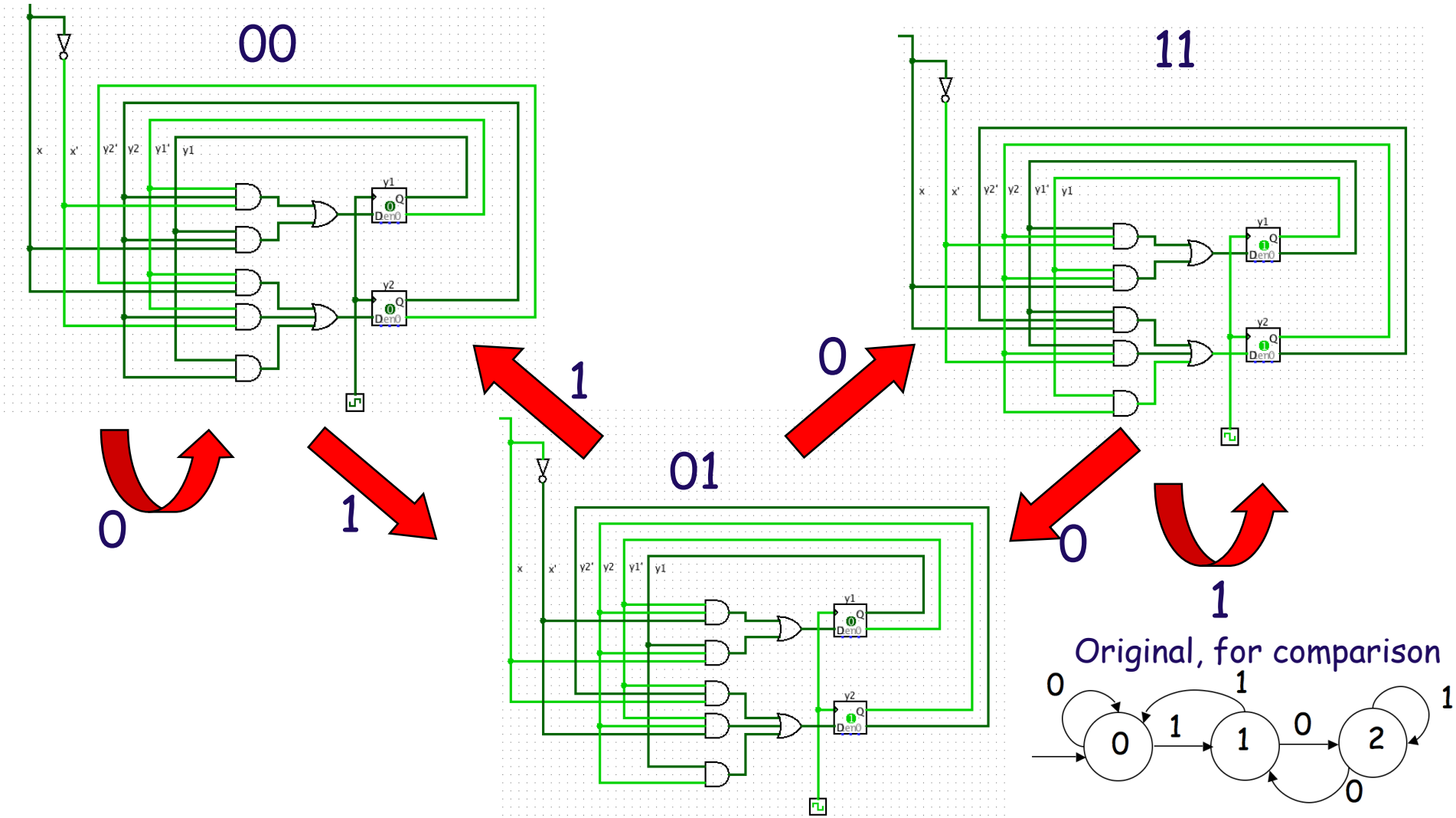
$$\text{next } y_2 = y_1'y_2'x + y_1'y_2x' + y_1y_2$$

Note that we need to use negations of flip-flop values, unlike with one-hot encoding. Also, we combined the last two minterms for the second function.

Design for Gray-Code Encoding



Check for Gray-Code Version Operation



Adding Output

- The preceding discussion only addresses the state-transition aspect of finite-state machines.
- To do useful work, our machine may need output as well.

Output Models

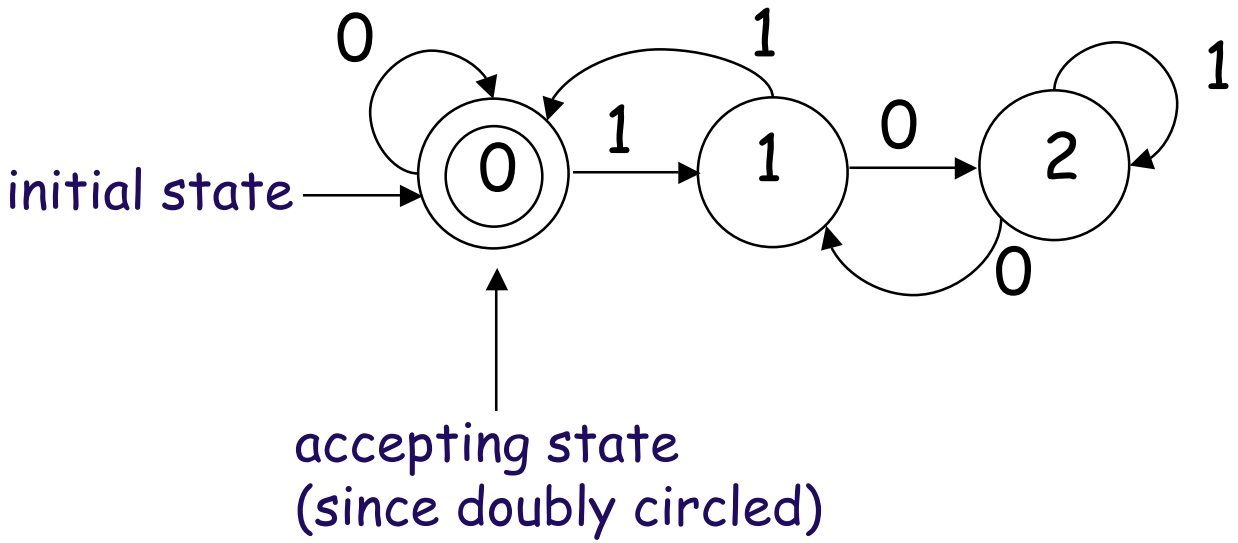
- There are two general models for output:
  - Attach output to transitions (called the **Mealy model**, after George H. Mealy)
  - Attach output to states (called the **Moore model**, after E.F. Moore)
    - A special case of just two possible output values is called an **acceptor**.

Acceptor

- Each state of an acceptor is either
  - Accepting, or
  - Rejecting
- The convention is that accepting states are doubly circled.
- There can be multiple accepting states.

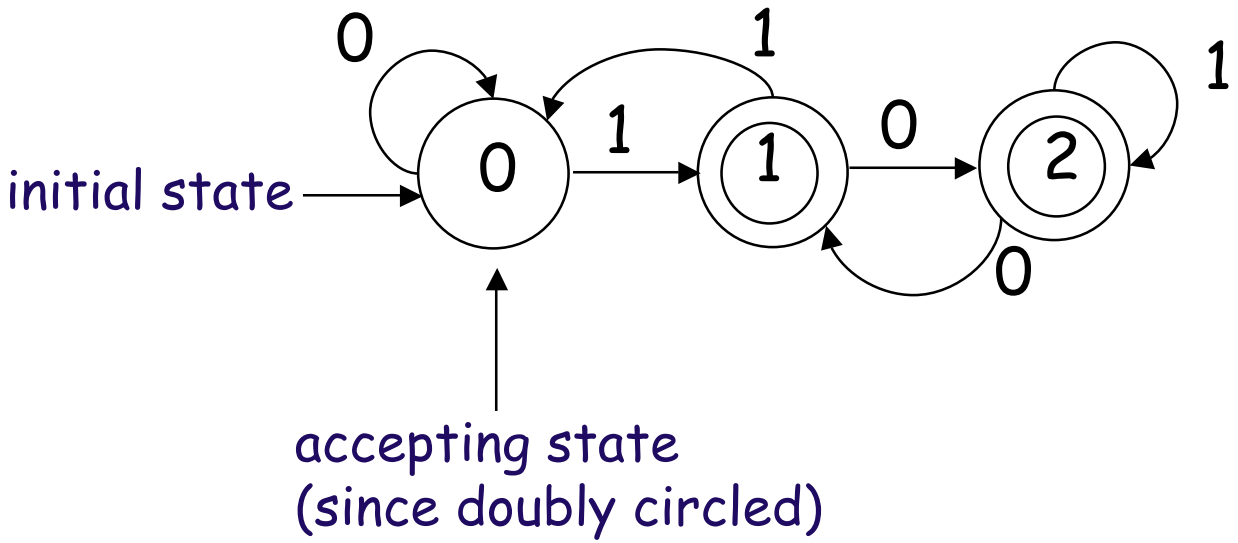
Acceptor Example

This acceptor accepts sequence that are multiples of 3 in binary, msb-first, e.g. 0, 11, 110, 1001, 1100, …



Acceptor Example 2

This acceptor accepts sequence that are *not* multiples of 3 in binary, msb-first, e.g. 1, 10, 100, 101, …



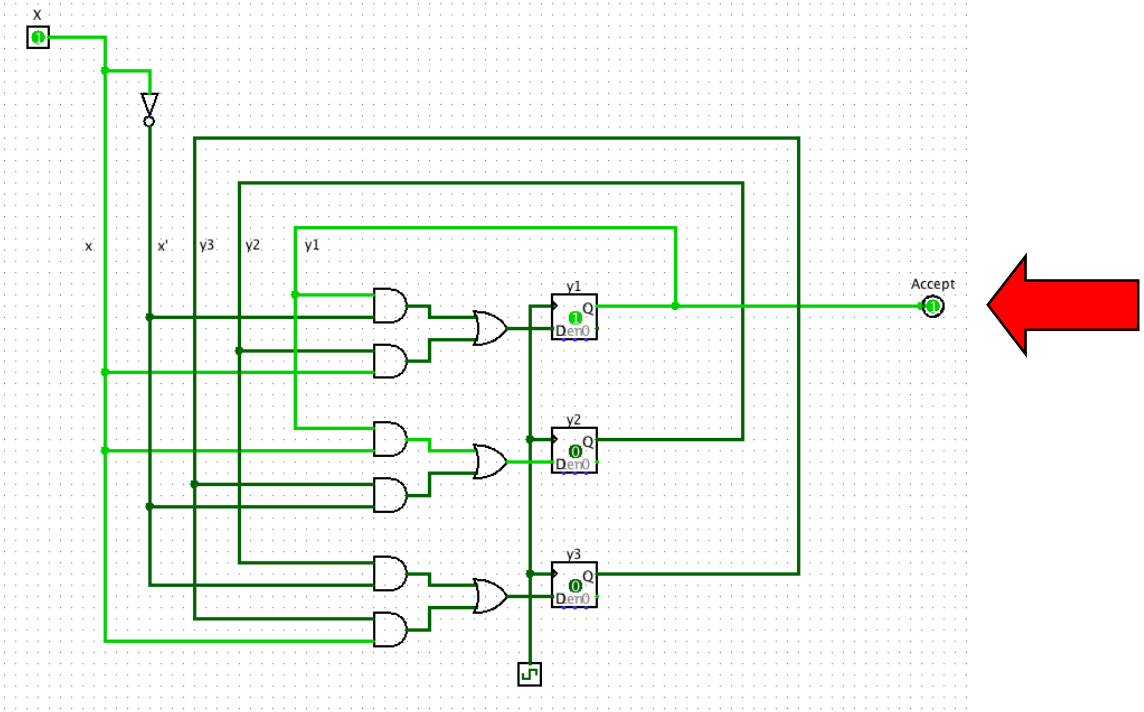
Note on Acceptance

- Acceptance and Rejection apply to the amount of the input sequence seen so far. They are generally not final conditions.

- It is possible to go from accept to reject and back, by providing more input.

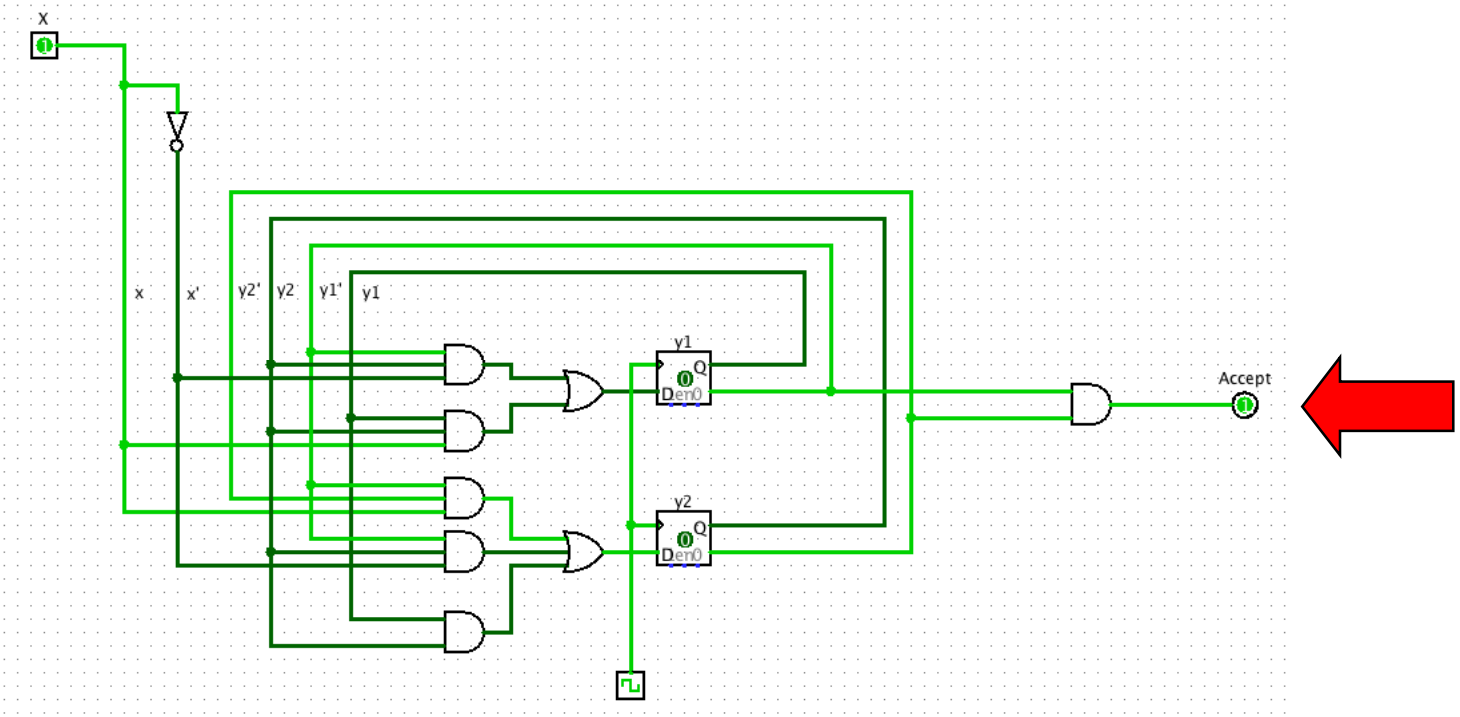
Acceptance → Output

- We can assign an output of 1 to acceptance, 0 to rejection using combinational logic. For multiple-of-3, one-hot:



Acceptance → Output

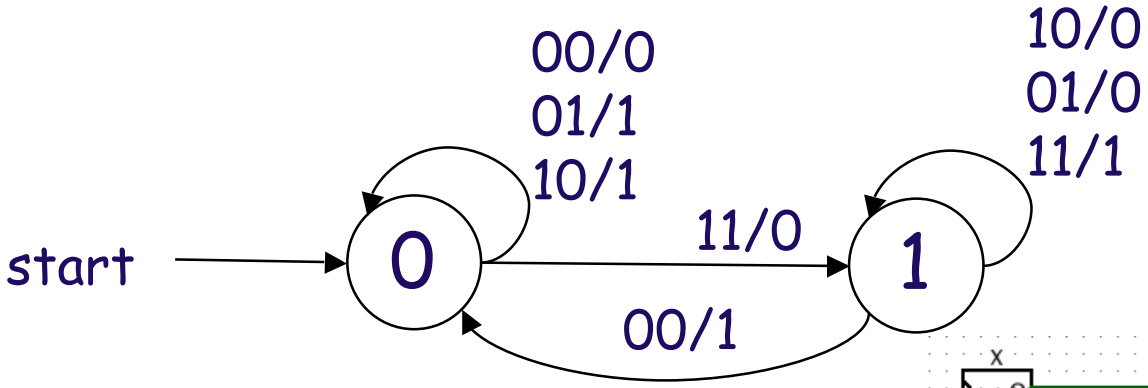
- 1 to acceptance, 0 to rejection using combinational logic. For multiple-of-3, Gray code:



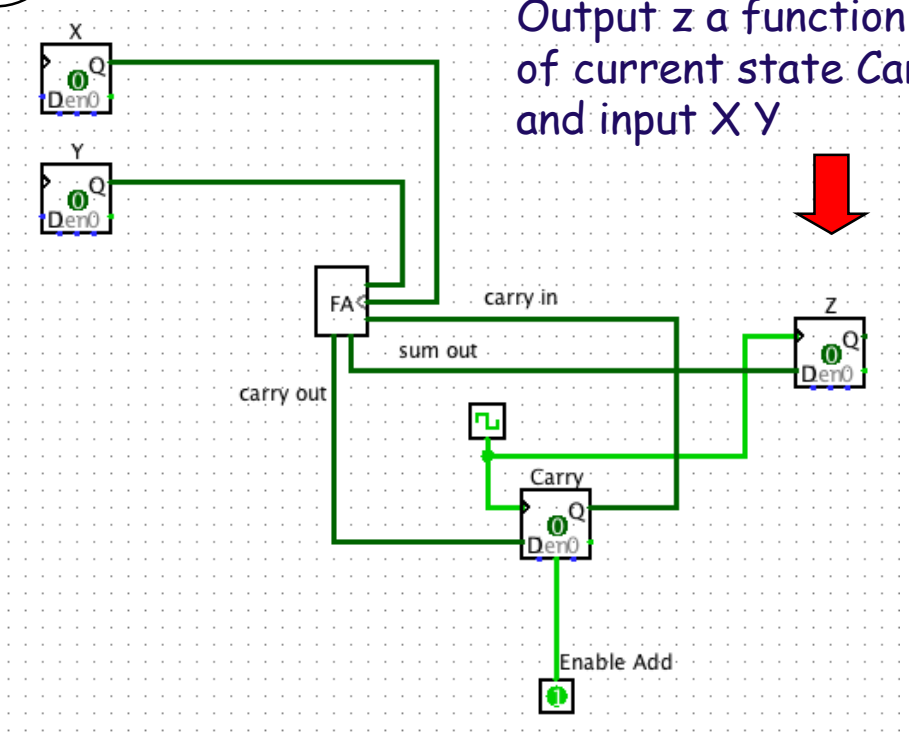
Mealy Model

- The output is attached to transitions, but a transition comes from a state and is based on an input.
- Therefore the output is a combination of the current state and the input.
- This output can be "captured" by flip-flops. This was indicated back on the sequential binary adder example.

Mealy Model Example



Output z a function of current state Carry and input X Y



Subsequent Discussions

- Finite-State Machines in the abstract
- Regular (type 3) languages: languages accepted by finite-state acceptors
- Regular expressions