Revised: 4/20/2020

Digital Logic III

Michael C. Hackett

Assistant Professor, Computer Science

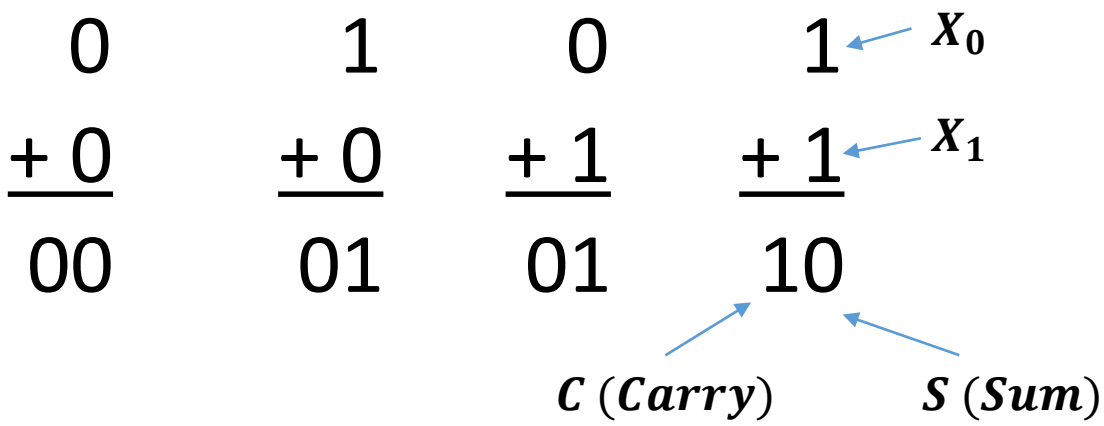
Community College of Philadelphia

Lecture Topics

- Combinational Circuits
  - Adders
    - Half Adder
    - Full Adder
  - Subtractors
    - Half Subtractor
    - Full Subtractor
  - Multipliers
- Sequential Circuits
  - Clocks
  - SR Latch
  - Flip-Flops
    - SR Flip-Flop
    - D Flip-Flop
    - JK Flip Flop
  - Registers

Adders

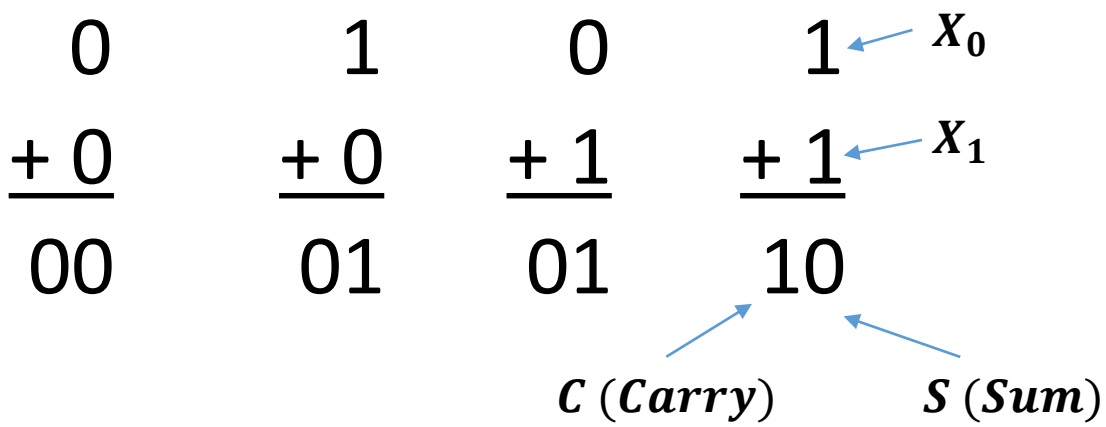
- **Adders** are combinational logic circuits capable of performing addition
- A **half adder** has two inputs (the two digits to add) and two outputs (the sum and the carry).



Half Adders

- Half adder truth table:

X_1	X_0	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Half Adders

SOP Expressions:

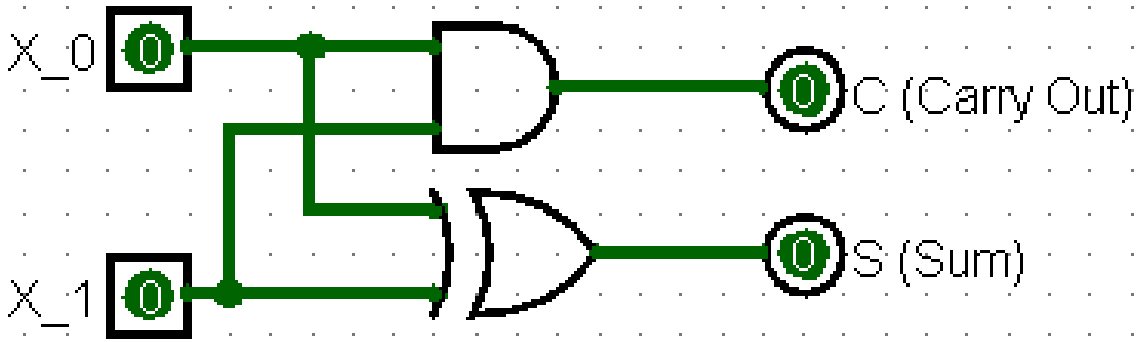
$$C = X_1 X_0$$

$$S = \overline{X_1} X_0 + X_1 \overline{X_0} = X_1 \oplus X_0$$

X_1	X_0	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half Adders

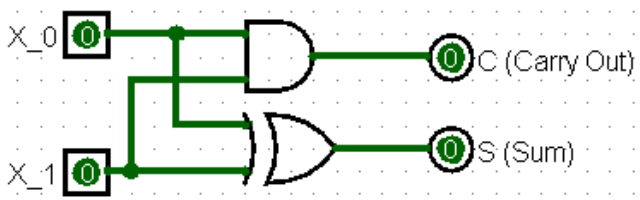
Half Adder Logic Circuit:



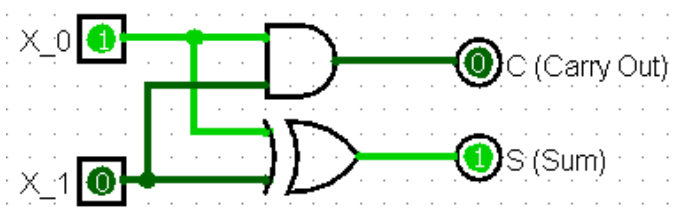
Half Adders

Half Adder Logic Circuit:

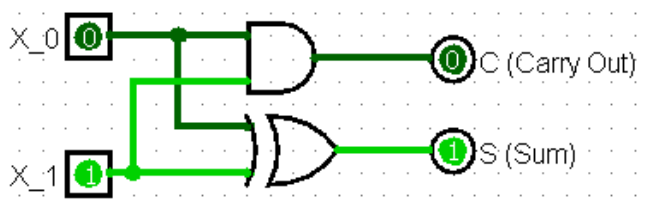
0
+ 0
00



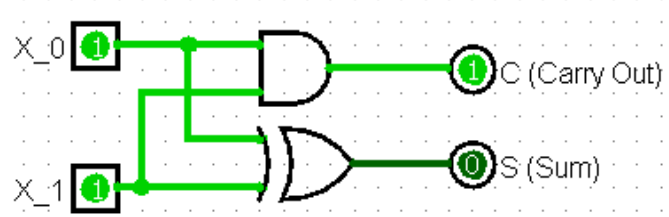
1
+ 0
01



0
+ 1
01

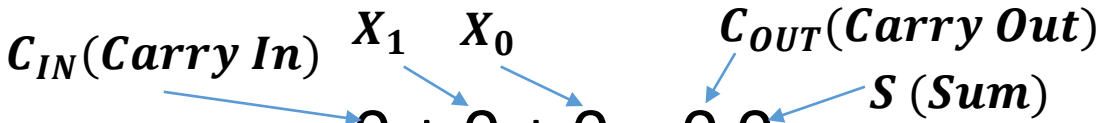


1
+ 1
10



Full Adders

- A **full adder** has three inputs (the two digits to add, plus a value *carried in*) and two outputs (the sum and the carry).



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

Full Adders

- Full adder truth table:

C_{IN}	X_1	X_0	C_{OUT}	S
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

Full Adders

SOP Expressions:

$$C_{OUT} = \overline{C_{IN}} X_1 X_0 + C_{IN} \overline{X_1} X_0 + C_{IN} X_1 \overline{X_0} + C_{IN} X_1 X_0$$

$$S = \overline{C_{IN}}\ \overline{X_1} X_0 + \overline{C_{IN}} X_1 \overline{X_0} + C_{IN} \overline{X_1}\ \overline{X_0} + C_{IN} X_1 X_0$$

C_{IN}	X_1	X_0	C_{OUT}	S
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

Full Adders

Simplifying:

$$C_{OUT} = \overline{\boldsymbol{C_{IN}}}\boldsymbol{X_1X_0} + C_{IN}\overline{X_1}X_0 + C_{IN}X_1\overline{X_0} + \boldsymbol{C_{IN}X_1X_0}$$

$$C_{OUT} = \boldsymbol{X_1X_0}(\overline{\boldsymbol{C_{IN}}} + \boldsymbol{C_{IN}}) + C_{IN}\overline{X_1}X_0 + C_{IN}X_1\overline{X_0}$$

$$C_{OUT} = \boldsymbol{X_1X_0}(\mathbf{1}) + C_{IN}\overline{X_1}X_0 + C_{IN}X_1\overline{X_0}$$

$$C_{OUT} = X_1X_0 + \boldsymbol{C_{IN}}\overline{\boldsymbol{X_1}}\boldsymbol{X_0} + \boldsymbol{C_{IN}X_1}\overline{\boldsymbol{X_0}}$$

$$C_{OUT} = X_1X_0 + C_{IN}(\overline{\boldsymbol{X_1}}\boldsymbol{X_0} + \boldsymbol{X_1}\overline{\boldsymbol{X_0}})$$

$$C_{OUT} = X_1X_0 + C_{IN}(X_0 \oplus X_1)$$

Full Adders

Simplifying:

$$S = \overline{C_{IN}}\,\overline{X_1}X_0 + \overline{C_{IN}}X_1\overline{X_0} + C_{IN}\overline{X_1}\,\overline{X_0} + C_{IN}X_1X_0$$

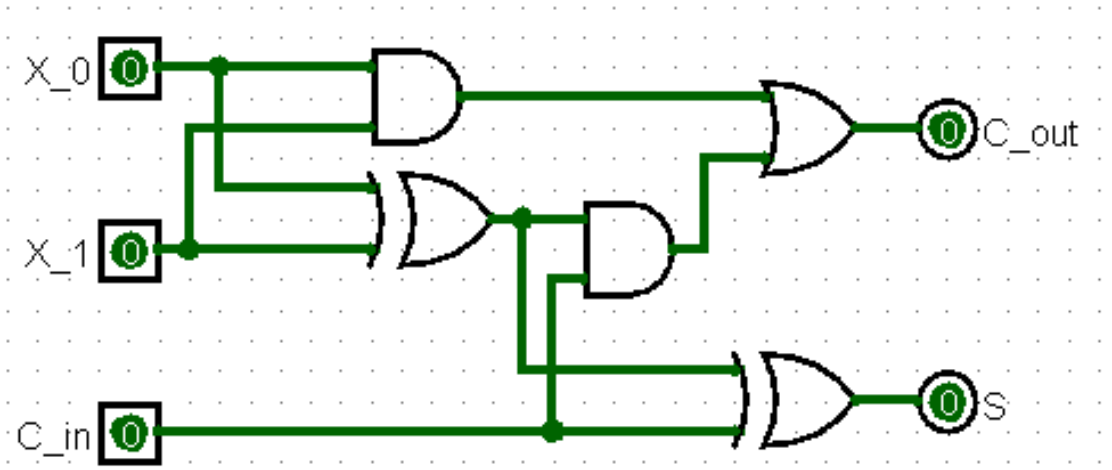
$$S = \overline{C_{IN}}\,(X_0 \oplus X_1) + C_{IN}(X_0 \odot X_1)$$

$$S = \overline{C_{IN}}\,(X_0 \oplus X_1) + C_{IN}(\overline{X_0 \oplus X_1}) \qquad \bar{X}Y + X\bar{Y} = X \oplus Y$$

$$S = C_{IN} \oplus (X_0 \oplus X_1) = C_{IN} \oplus X_0 \oplus X_1$$

Full Adders

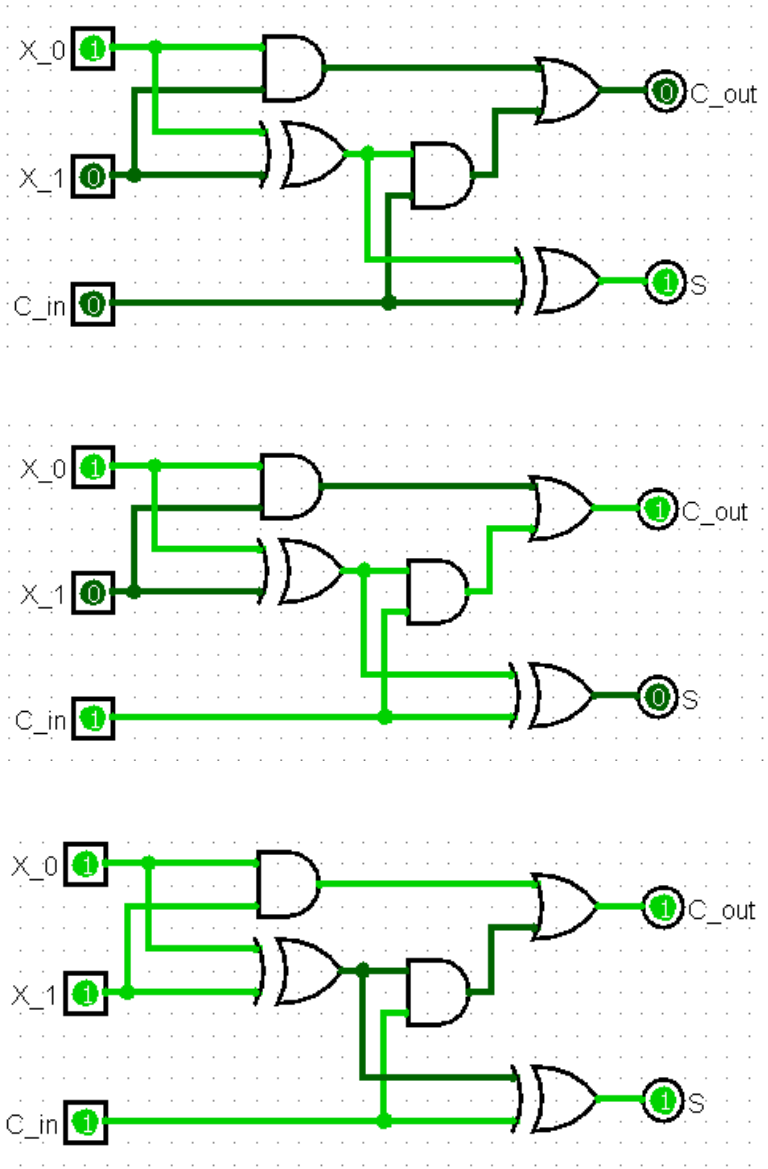
Full Adder Logic Circuit:



Full Adders

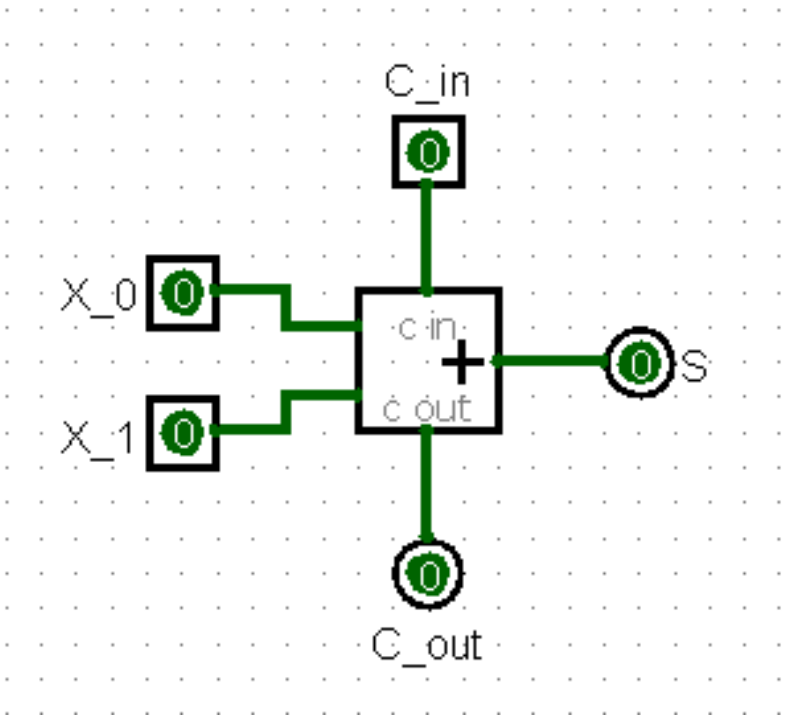
Full Adder Logic Circuit:

C_{IN}	X_1	X_0	C_{OUT}	S
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**



Full Adders

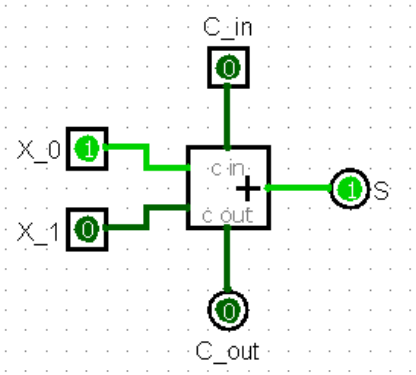
- Abstracted Full Adder:

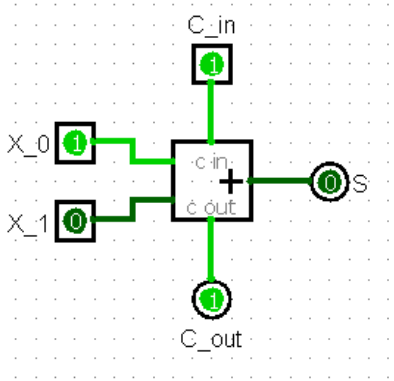


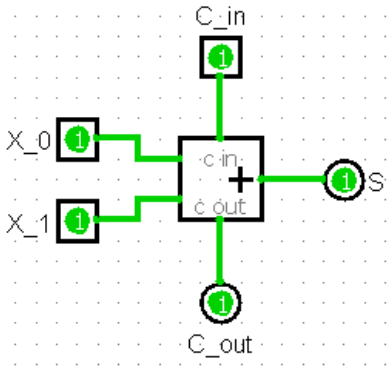
Full Adders

- Abstracted Full Adder:

C_{IN}	X_1	X_0	C_{OUT}	S
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**







Full Adders

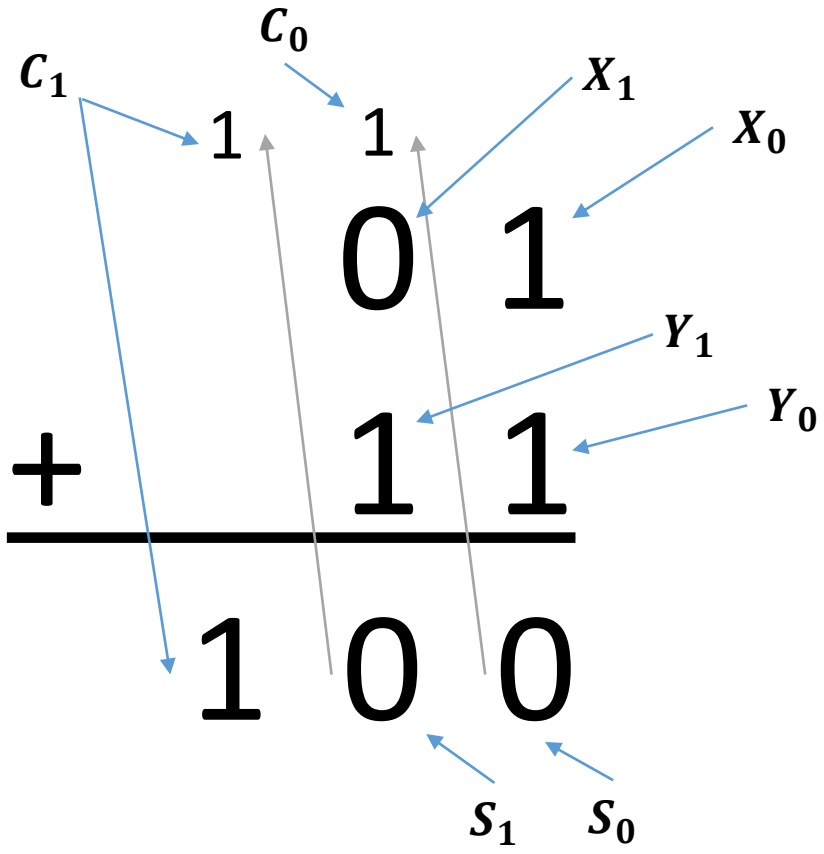
- So far, we've seen only how adders can add single digits together
  - 1+1+1 or 0+1 or 1+0+1 is no problem
  - What if we wanted to add 10+11 or 11101+10101?

- Full adders can work together by providing the carry out of one full adder as the carry in for a second adder
  - The technique shown next is a *ripple-carry adder*

$$\begin{array}{r} 0\ 0\ 1 \\ +\ 0\ 1\ 1 \\ \hline 1\ 0\ 0 \end{array}$$

Full Adders

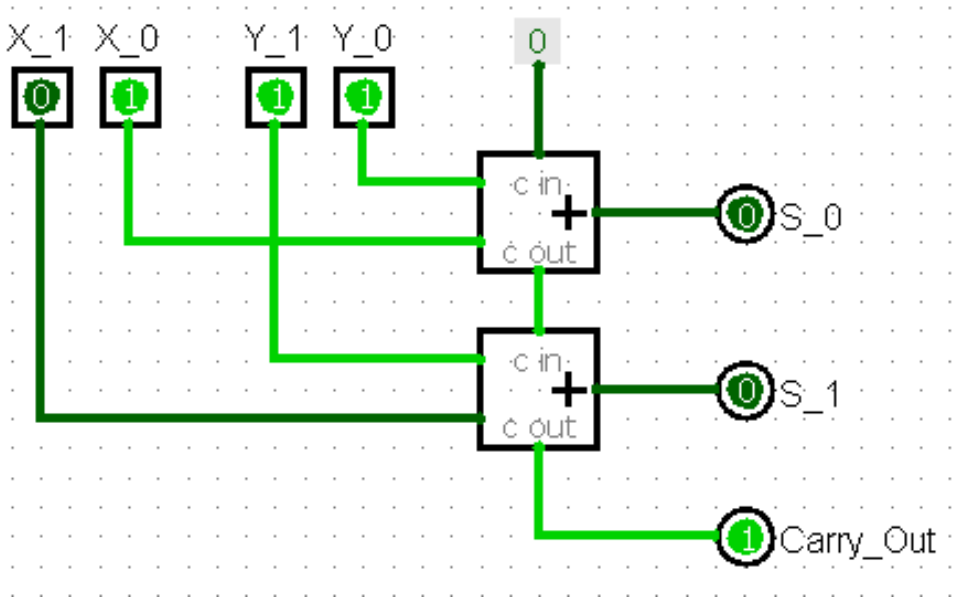
- For example:



Full Adders

- A 2-bit Adder:

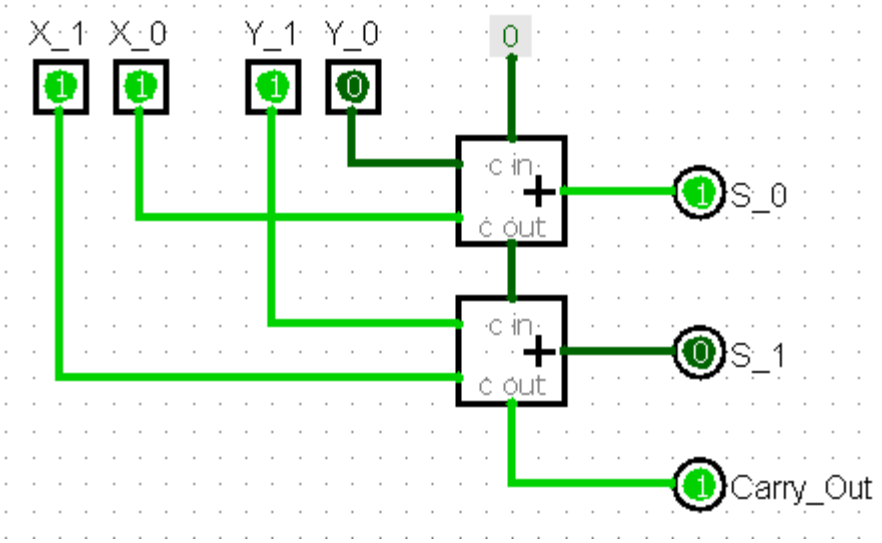
```
    0 1
+   1 1
-------
  1 0 0
```



Full Adders

- Another example:

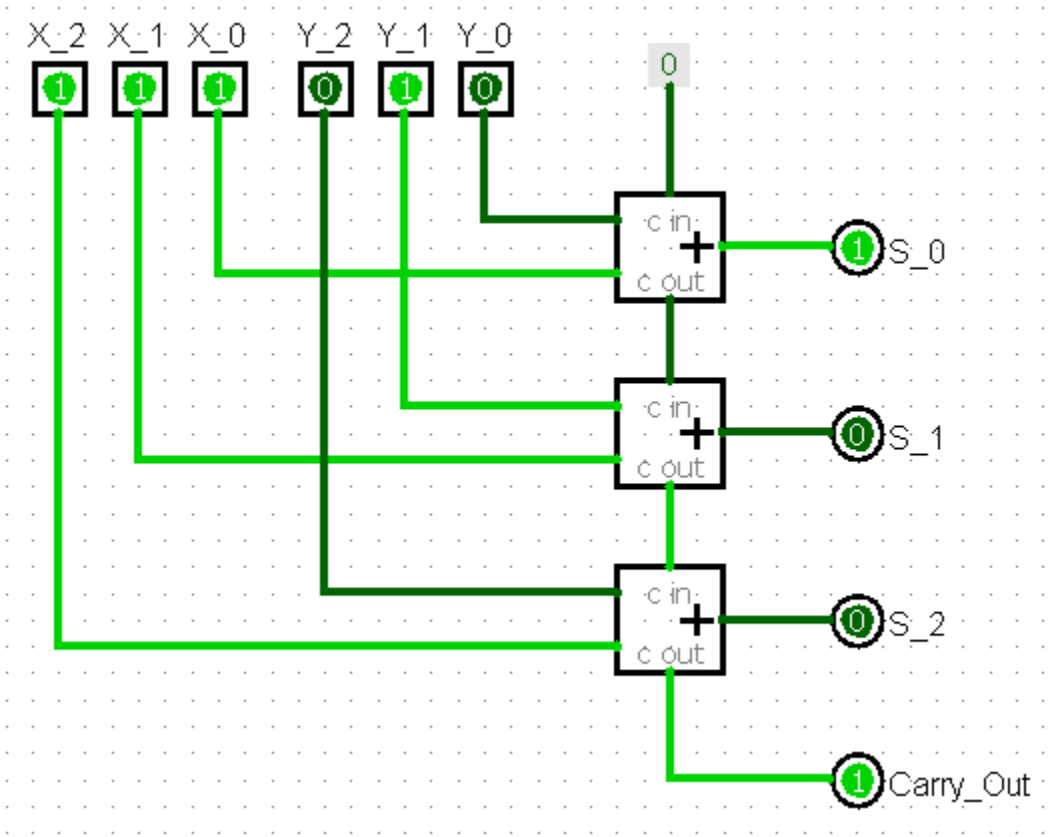
$$\begin{array}{r} 1\ 1 \\ +\quad 1\ 0 \\ \hline 1\ 0\ 1 \end{array}$$



Full Adders

- A 3-bit Adder:

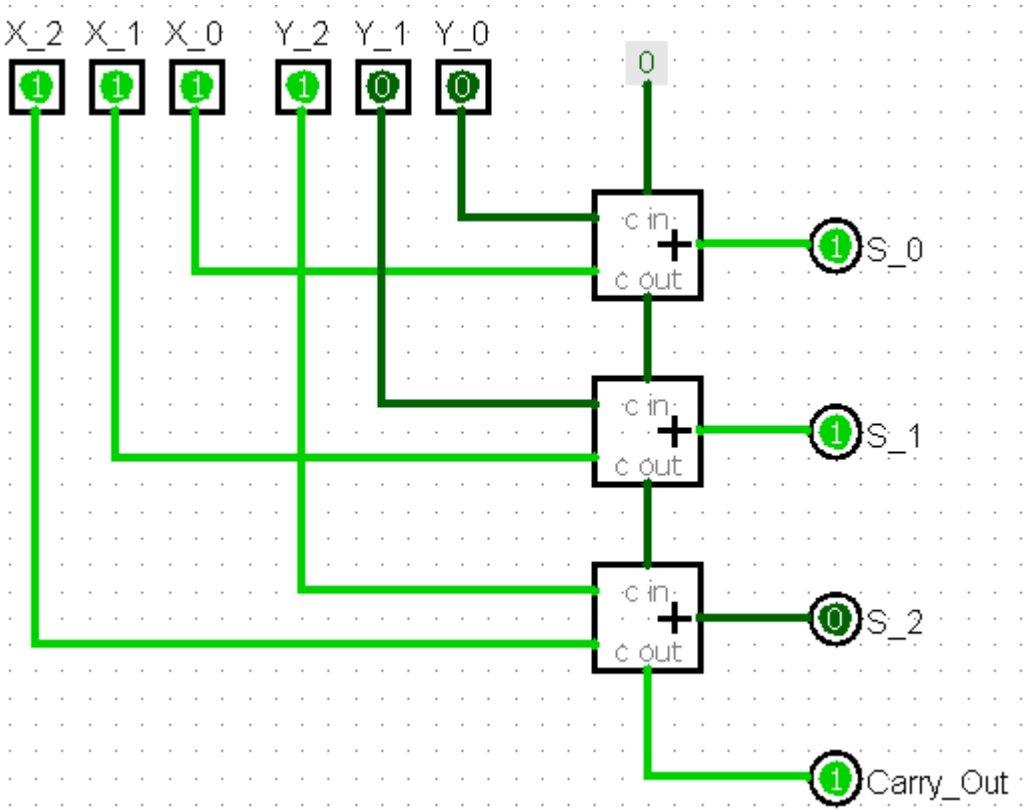
$$\begin{array}{r} 1\ 1\ 1 \\ +\ 0\ 1\ 0 \\ \hline 1\ 0\ 0\ 1 \end{array}$$



Full Adders

- Another example:

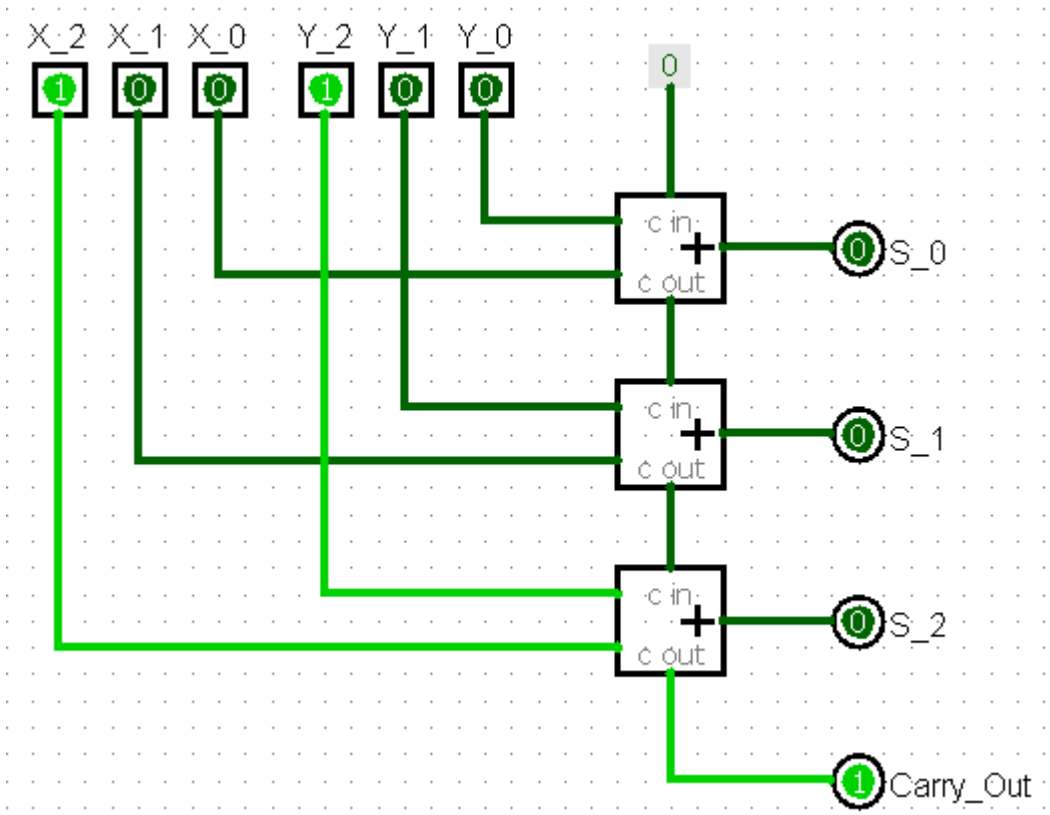
$$\begin{array}{r} 1\ 1\ 1 \\ +\quad 1\ 0\ 0 \\ \hline 1\ 0\ 1\ 1 \end{array}$$



Full Adders

- Another example:

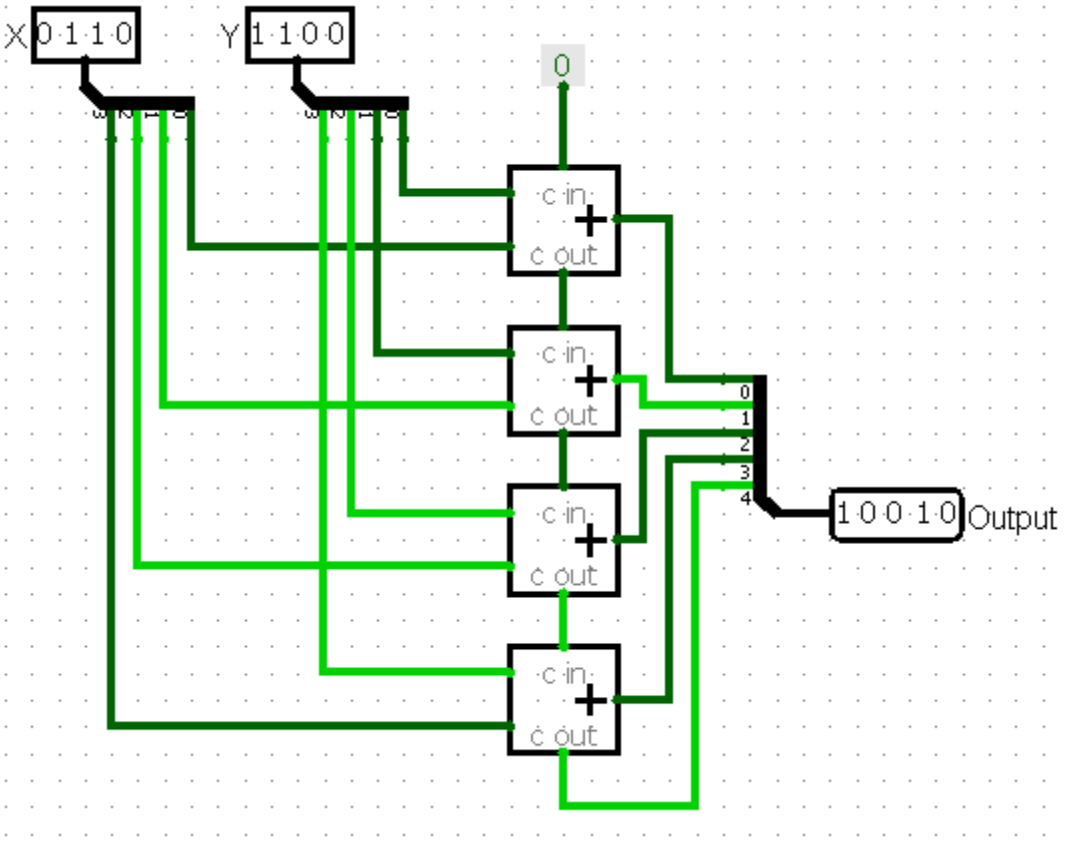
$$\begin{array}{r} 1\ 0\ 0 \\ +\ 1\ 0\ 0 \\ \hline 1\ 0\ 0\ 0 \end{array}$$



Full Adders

- A 4-bit Adder (with buses):

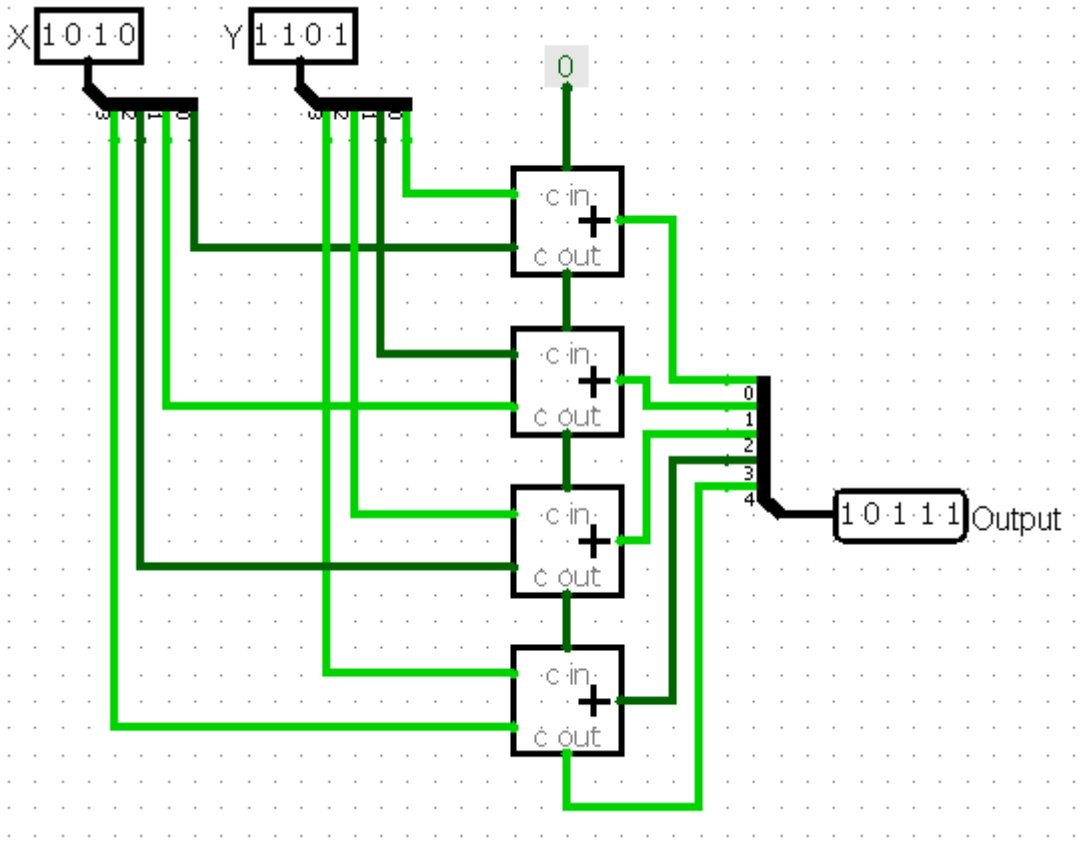
$$\begin{array}{r} 0\ 1\ 1\ 0 \\ +\ 1\ 1\ 0\ 0 \\ \hline 1\ 0\ 0\ 1\ 0 \end{array}$$



Full Adders

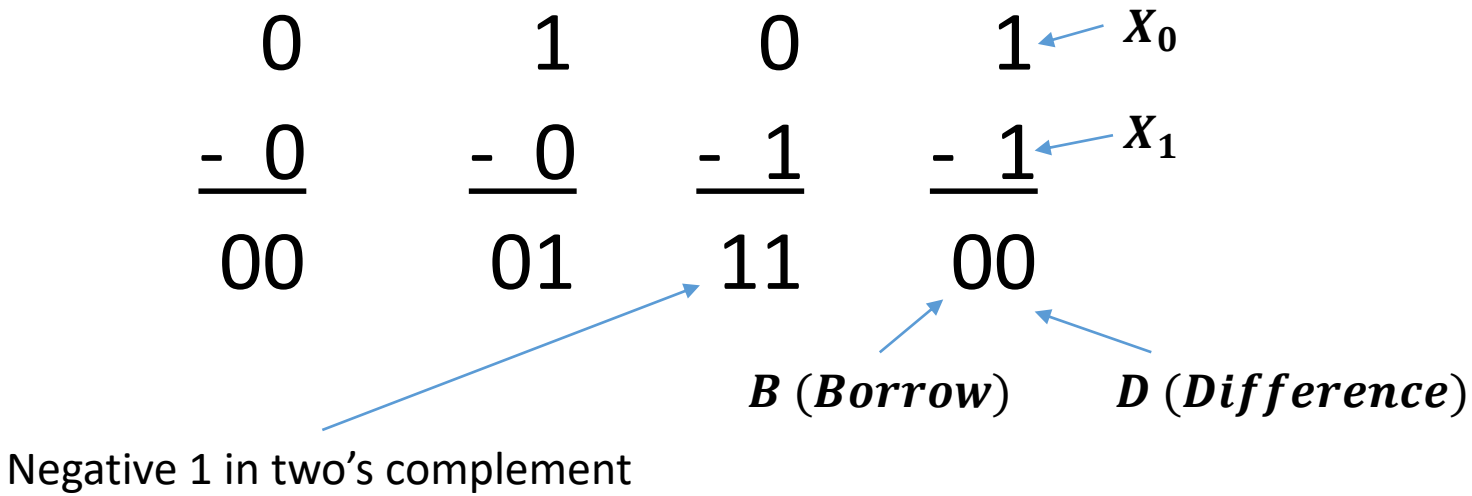
- Another example

$$
\begin{array}{r}
1\ 0\ 1\ 0 \\
+\ 1\ 1\ 0\ 1 \\
\hline
1\ 0\ 1\ 1\ 1
\end{array}
$$



Subtractors

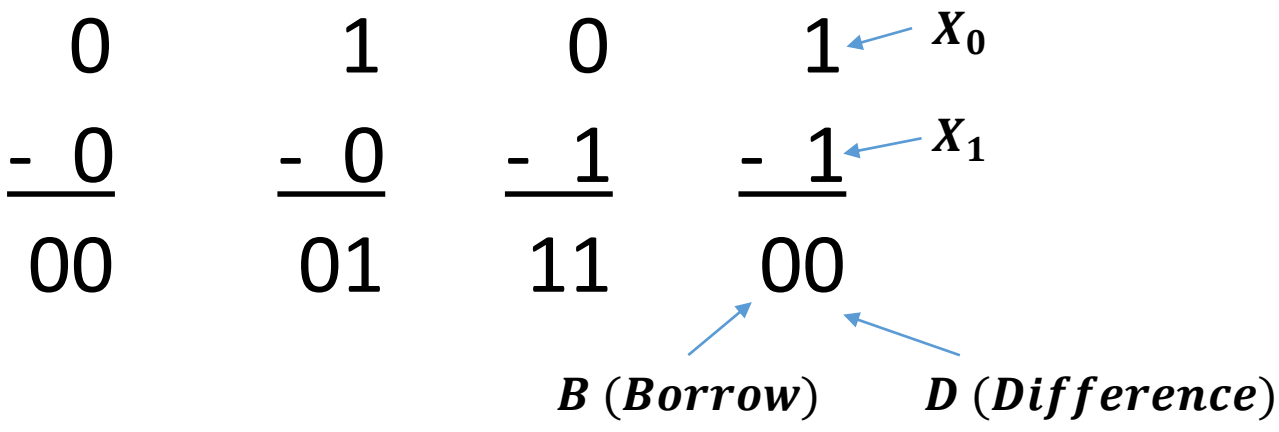
- **Subtractors** are combinational logic circuits capable of performing subtraction
- A **half subtractor** has two inputs (the two digits to subtract) and two outputs (the difference and the borrow).



Half Subtractor

- Half subtractor truth table:

X_1	X_0	B	D
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	0



Half Subtractors

SOP Expressions:

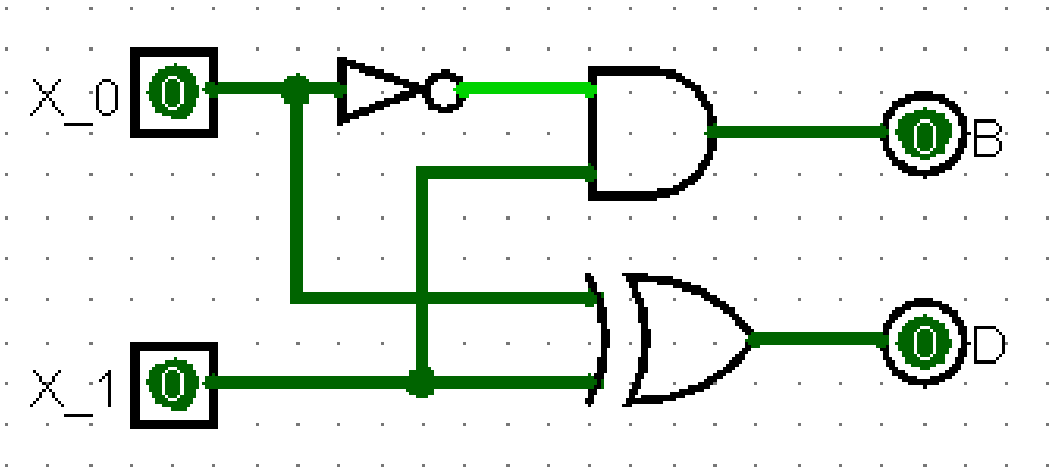
$$B = X_1\overline{X_0}$$

$$D = \overline{X_1}X_0 + X_1\overline{X_0} = X_1 \oplus X_0$$

X_1	X_0	B	D
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	0

Half Subtractor

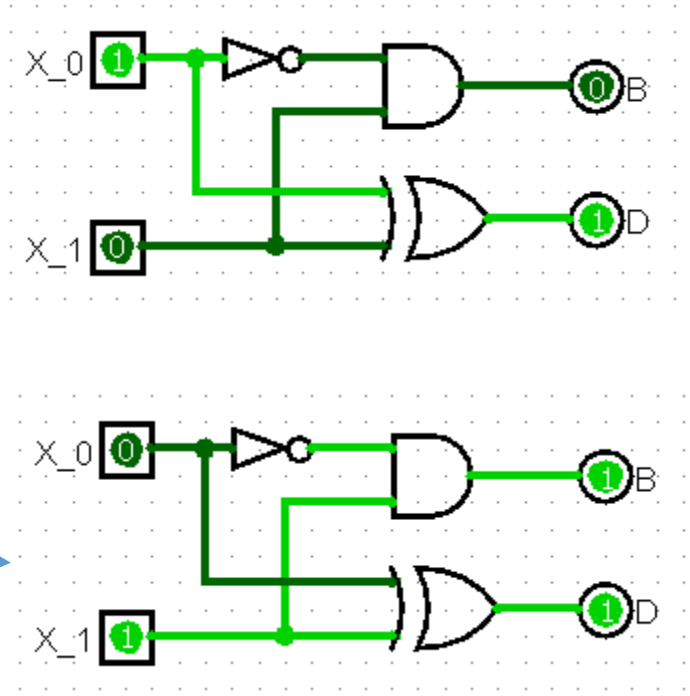
Half Subtractor Logic Circuit:



Half Subtractor

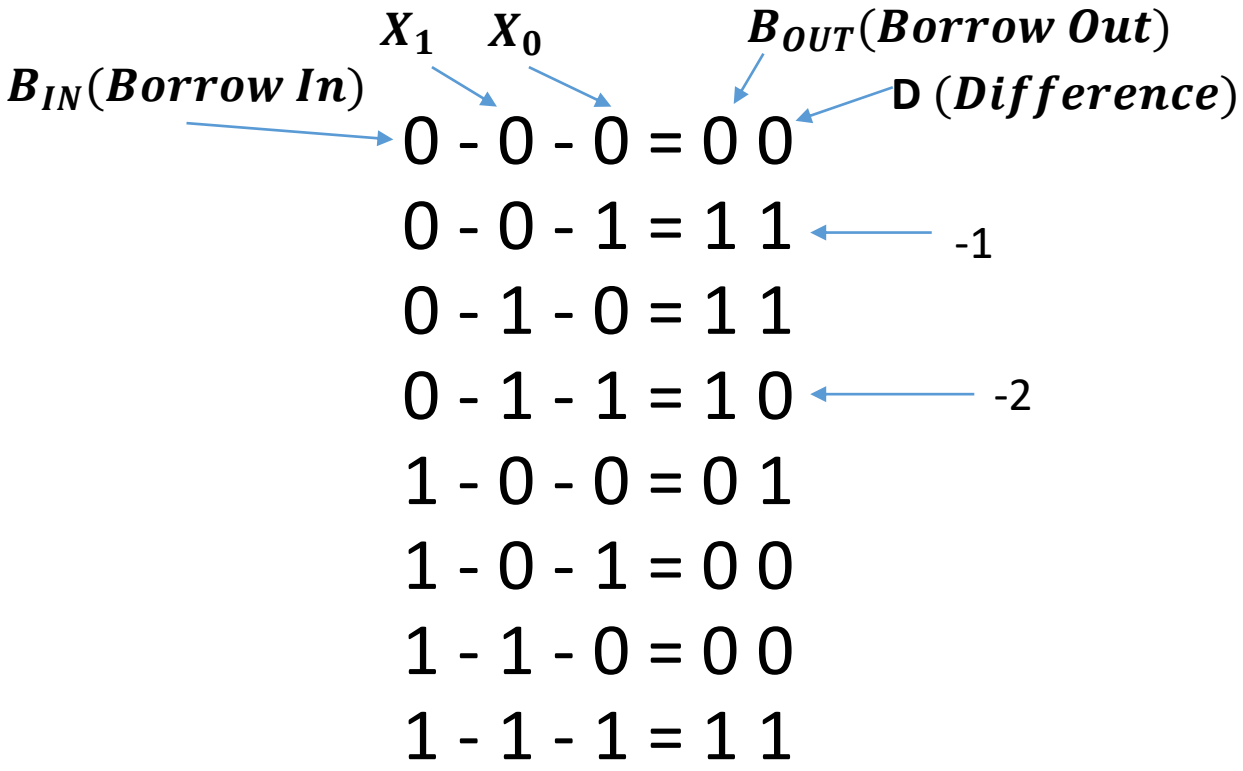
Half Subtractor Logic Circuit:

X_1	X_0	B	D
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	0



Full Subtractors

- A **full subtractor** has three inputs (the two digits to subtract, plus a value *borrowed in*) and two outputs (the different and the borrow).



Full Subtractors

SOP Expressions:

$$B_{OUT} = \overline{B_{IN}}\,\overline{X_1}X_0 + \overline{B_{IN}}X_1\overline{X_0} + \overline{B_{IN}}X_1X_0 + B_{IN}X_1X_0$$

$$D = \overline{B_{IN}}\,\overline{X_1}X_0 + \overline{B_{IN}}X_1\overline{X_0} + B_{IN}\overline{X_1}\,\overline{X_0} + B_{IN}X_1X_0$$

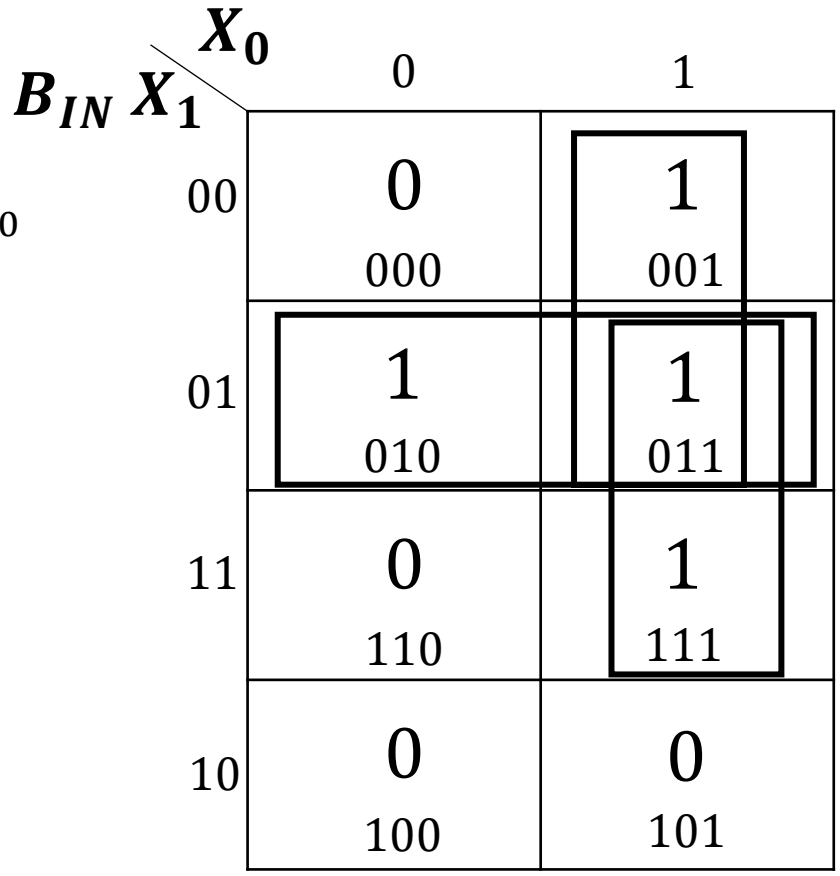
B_{IN}	X_1	X_0	B_{OUT}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Full Subtractors

Simplifying:

$$B_{OUT} = \overline{B_{IN}}\,\overline{X_1}X_0 + \overline{B_{IN}}X_1\overline{X_0} + \overline{B_{IN}}X_1X_0 + B_{IN}X_1X_0$$

$$B_{OUT} = \overline{B_{IN}}X_1 + \overline{B_{IN}}X_0 + X_1X_0$$



Full Subtractors

Simplifying:

$$D = \overline{B_{IN}}\,\overline{X_1}X_0 + \overline{B_{IN}}X_1\overline{X_0} + B_{IN}\overline{X_1}\,\overline{X_0} + B_{IN}X_1X_0$$

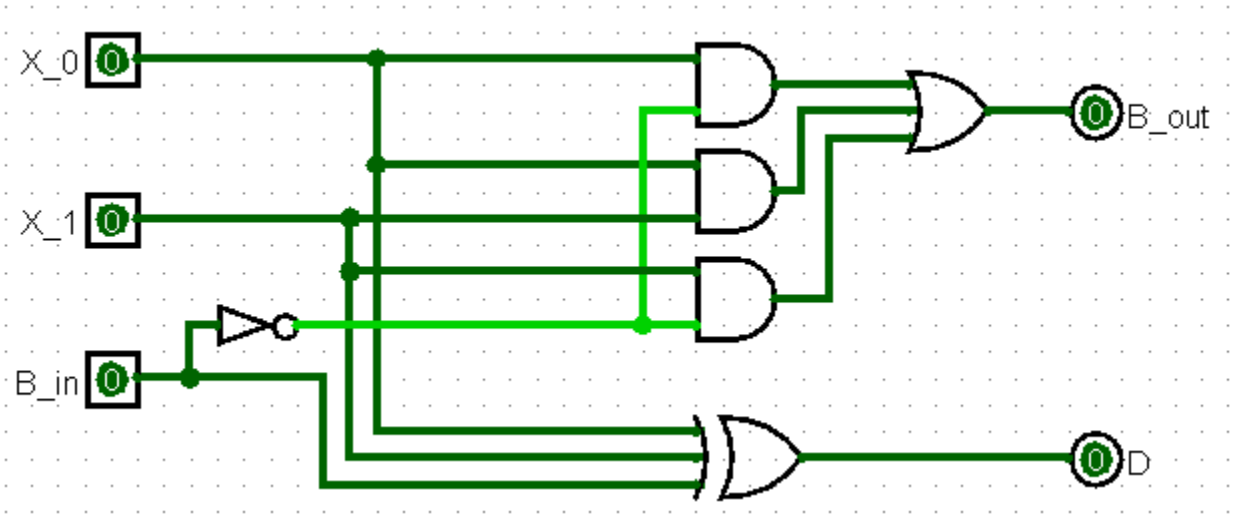
$$D = \overline{B_{IN}}(X_0 \oplus X_1) + B_{IN}(X_0 \odot X_1)$$

$$D = \overline{B_{IN}}(X_0 \oplus X_1) + B_{IN}(\overline{X_0 \oplus X_1}) \qquad \bar{X}Y + X\bar{Y} = X \oplus Y$$

$$D = B_{IN} \oplus (X_0 \oplus X_1) = B_{IN} \oplus X_0 \oplus X_1$$

Full Subtractor

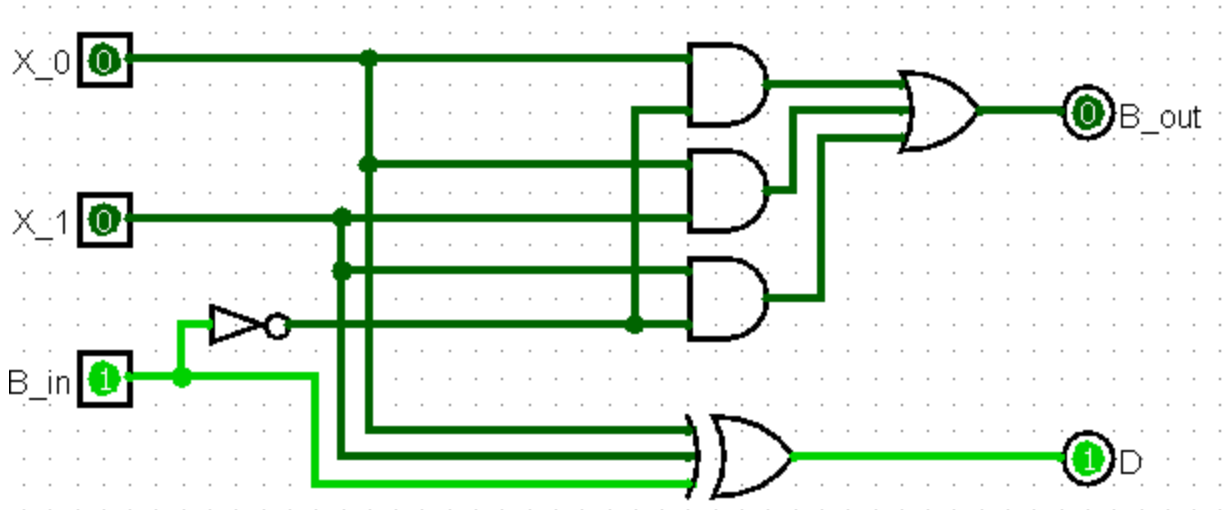
Full Subtractor Logic Circuit:



Full Subtractor

Full Subtractor Logic Circuit:

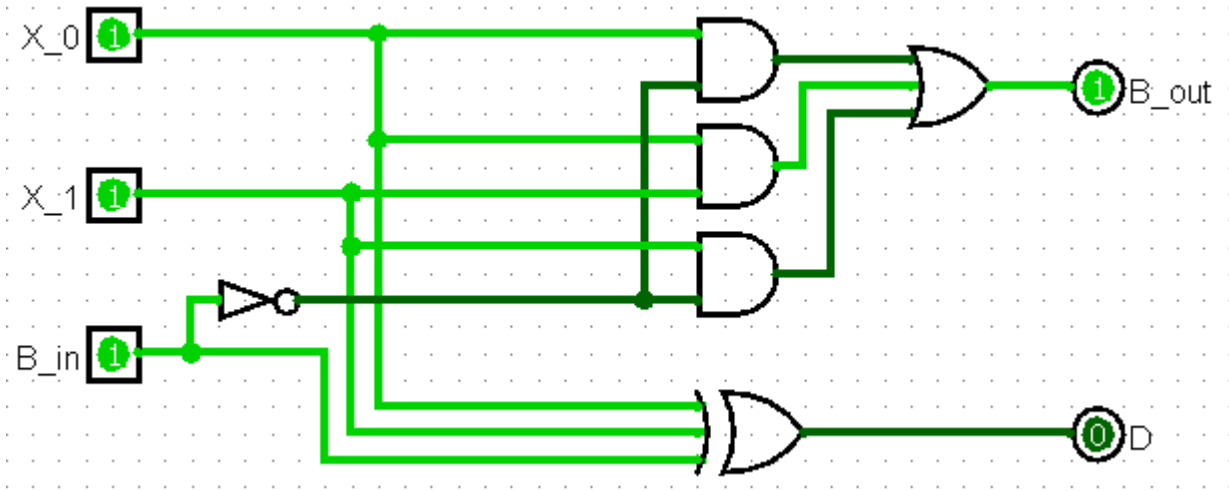
B_{IN}	X_1	X_0	B_{OUT}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	**0**	**0**	**0**	**1**
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



Full Subtractor

Full Subtractor Logic Circuit:

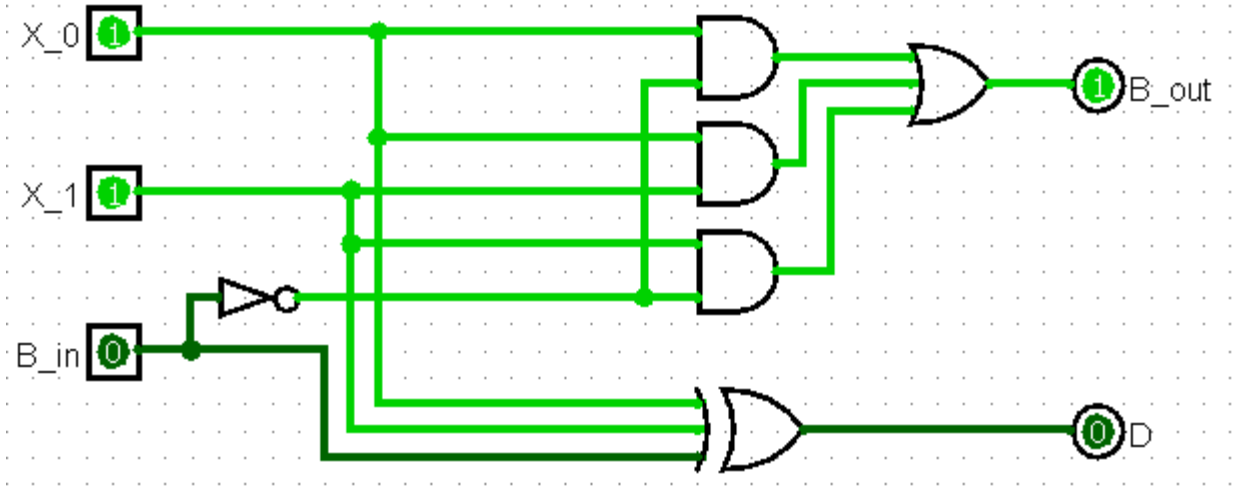
B_{IN}	X_1	X_0	B_{OUT}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	**1**	**1**	**1**	**1**



Full Subtractor

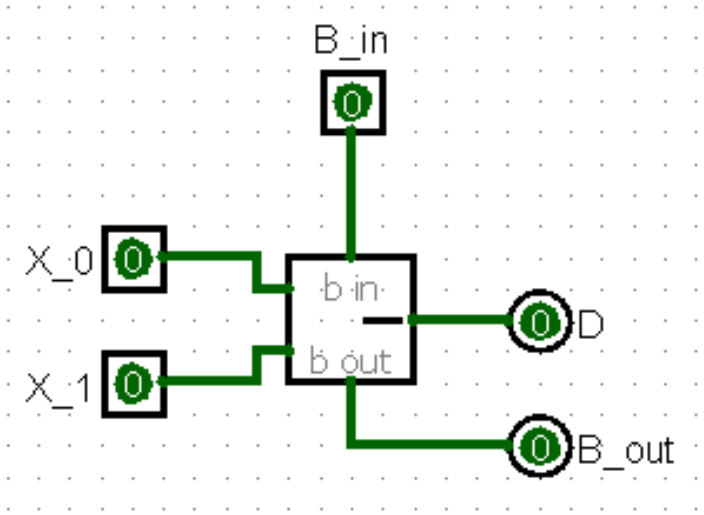
Full Subtractor Logic Circuit:

B_{IN}	X_1	X_0	B_{OUT}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	**1**	**1**	**1**	**0**
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



Full Subtractors

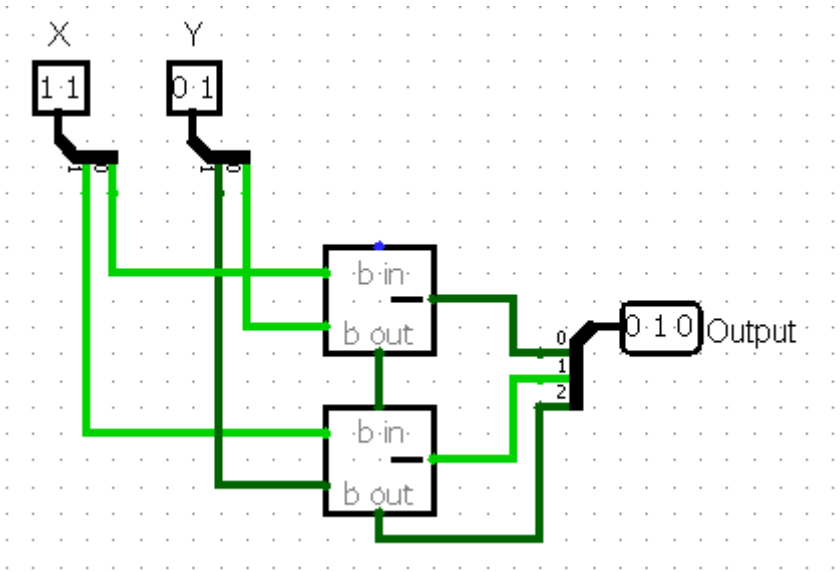
- Abstracted Full Subtractor:



Full Subtractors

- Like full adders, full subtractors can work together by providing the borrow out of one full subtractor as the borrow in for a second subtractor

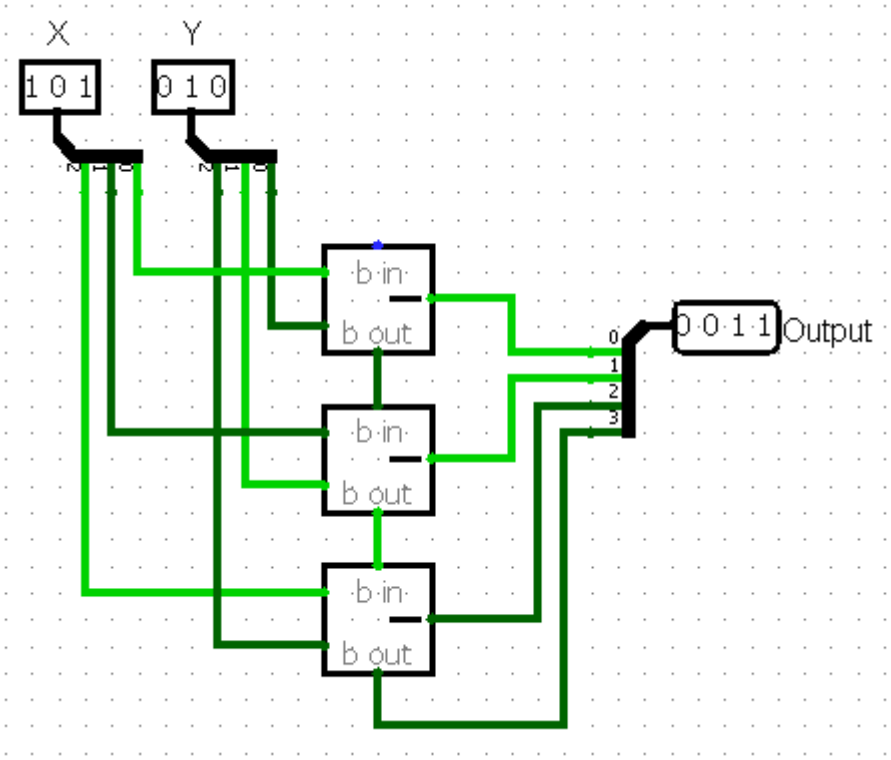
$$\begin{array}{r} 1\ 1 \\ -\quad 0\ 1 \\ \hline 0\ 1\ 0 \end{array}$$



Full Subtractors

- A 3-bit Subtractor:

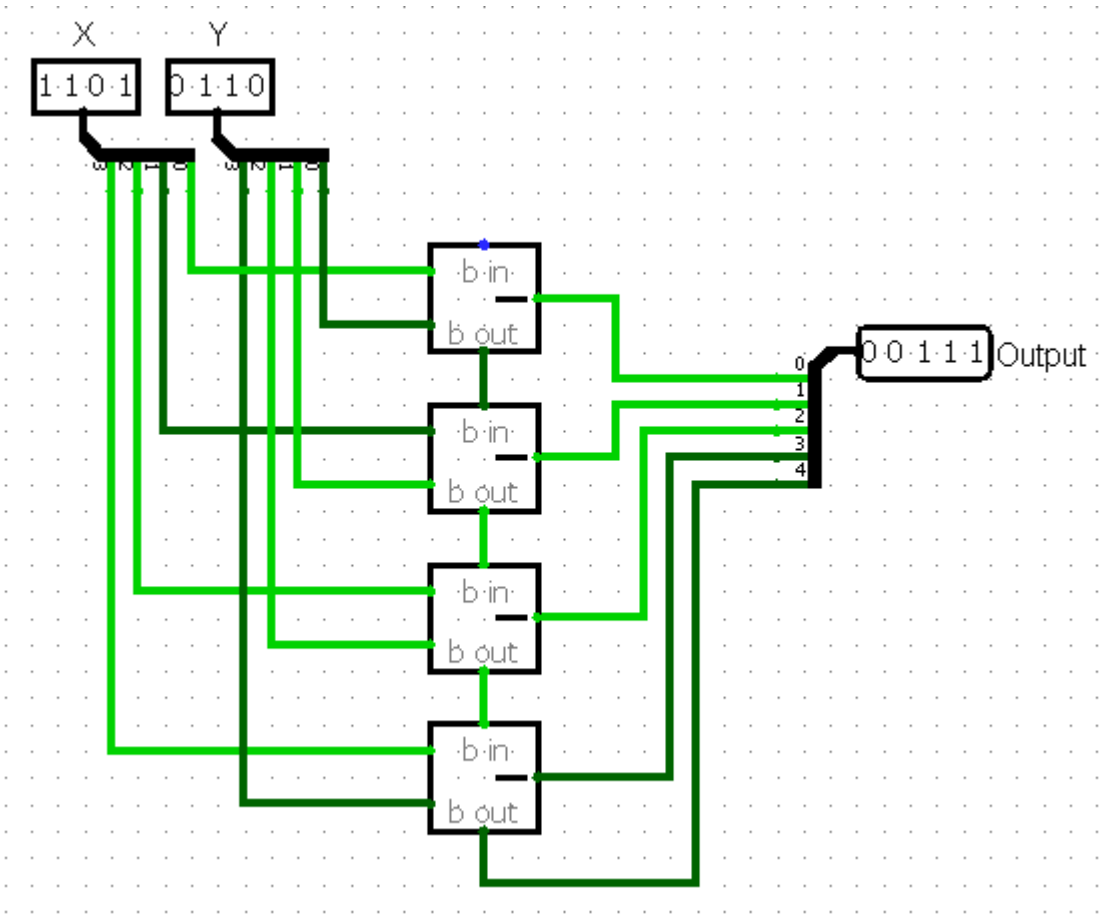
$$\begin{array}{r} 1\ 0\ 1 \\ -\ 0\ 1\ 0 \\ \hline 0\ 0\ 1\ 1 \end{array}$$



Full Subtractors

- A 4-bit Subtractor:

$$\begin{array}{r} 1\ 1\ 0\ 1 \\ -\ 0\ 1\ 1\ 0 \\ \hline 0\ 0\ 1\ 1\ 1 \end{array}$$



Multipliers

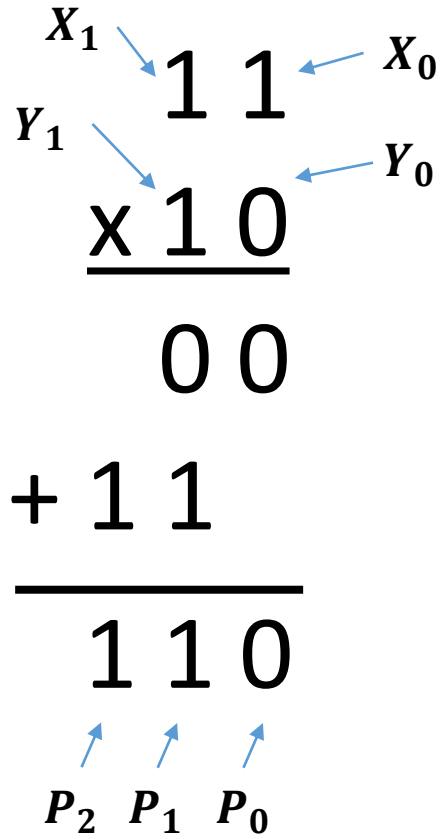
- **Multipliers** (not to be confused with multiplexers) are combinational logic circuits capable of performing multiplication
- Note that the multiplication of two 1-bit numbers is a simple *and* operation

0	1	0	1	← X_0
x 0	x 0	x 1	x 1	← Y_0
0	0	0	1	

X_0	Y_0	$X_0 \times Y_0$	$X_0 \cdot Y_0$
0	0	**0**	**0**
0	1	**0**	**0**
1	0	**0**	**0**
1	1	**1**	**0**

Multipliers

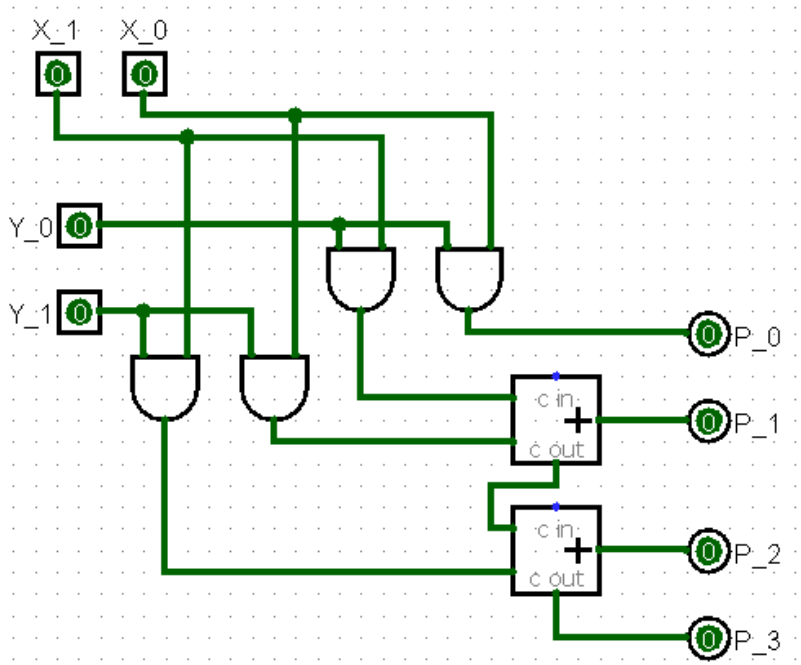
- However, addition will be required when multiplying numbers that are two or more bits.
- We will see how to construct multipliers using full and half adders.
- The largest product of multiplying two, 2-bit numbers is 9:
  - 11 x 11 = 1001 (3 x 3 = 9)
  - Thus, our circuit must have 4 outputs
    - P_0 through P_3



Multipliers

2-bit Multiplier Logic Circuit:

- (Uses 2 half adders)



Multipliers

2-bit Multiplier Logic Circuit:

X_1 1 1 X_0

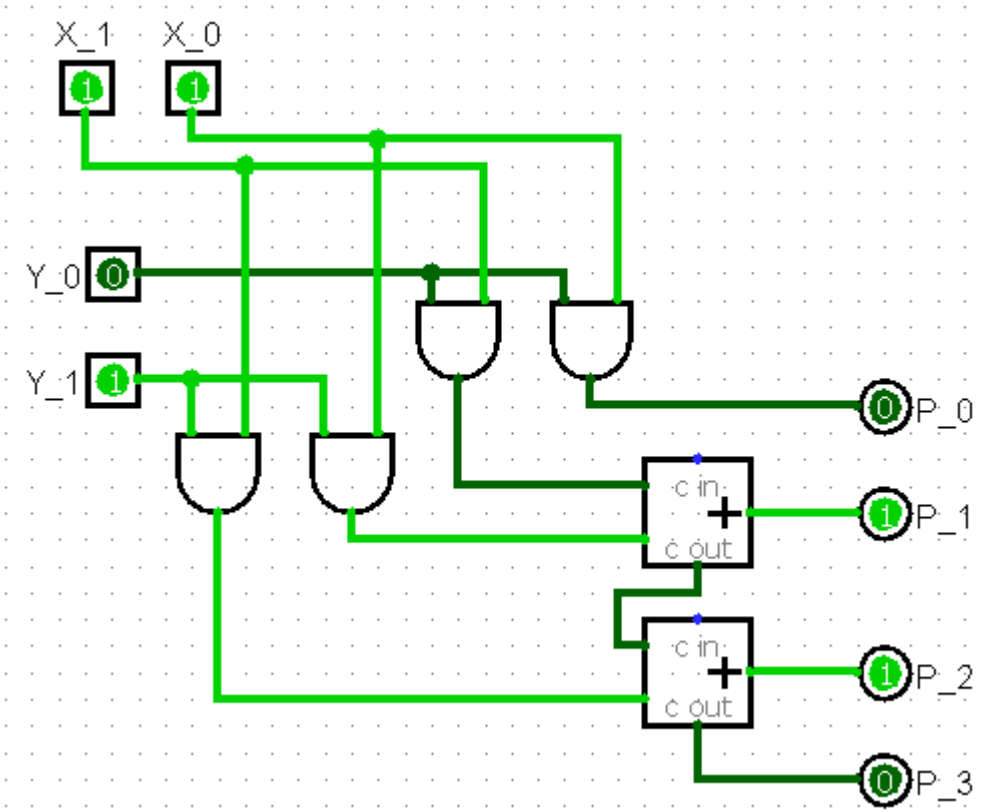
Y_1 x 1 0 Y_0

0 0

+ 1 1

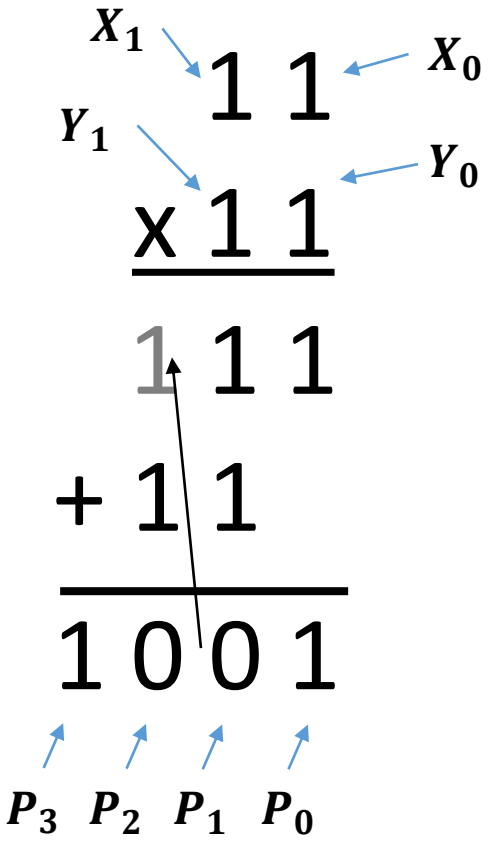
1 1 0

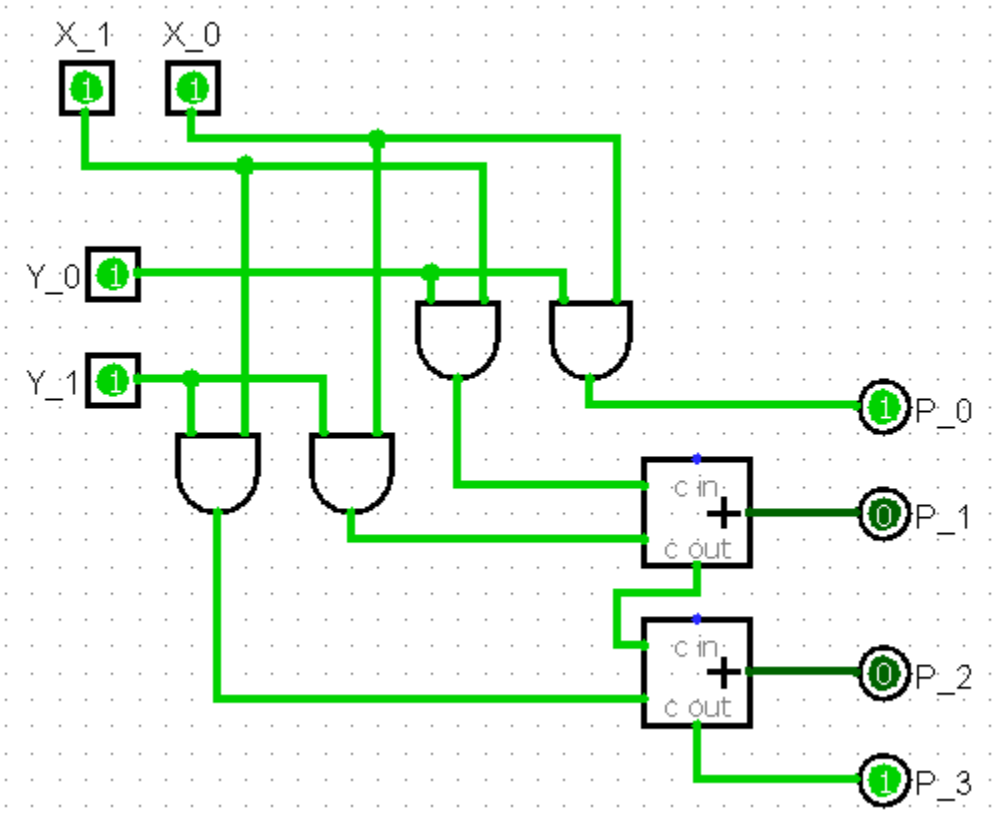
P_2 P_1 P_0



Multipliers

2-bit Multiplier Logic Circuit:

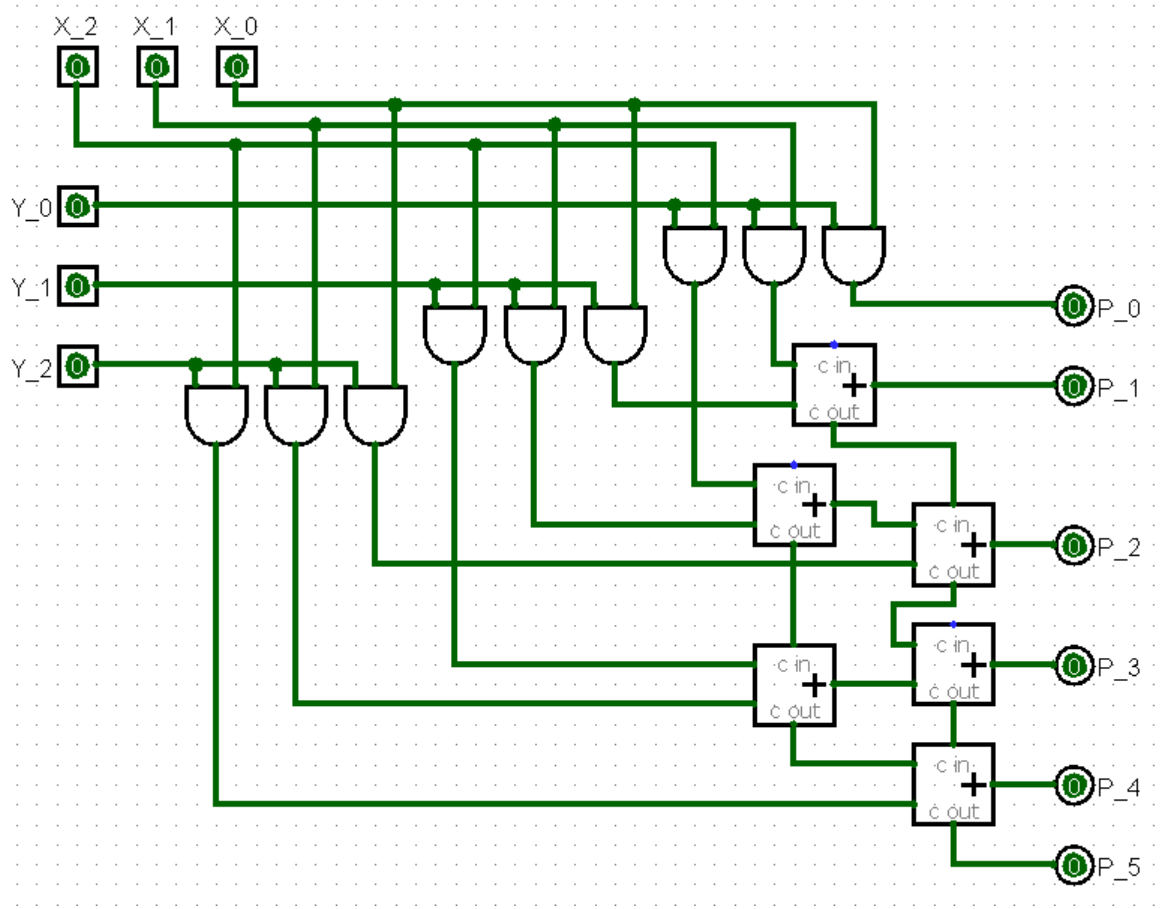




Multipliers

3-bit Multiplier Logic Circuit:

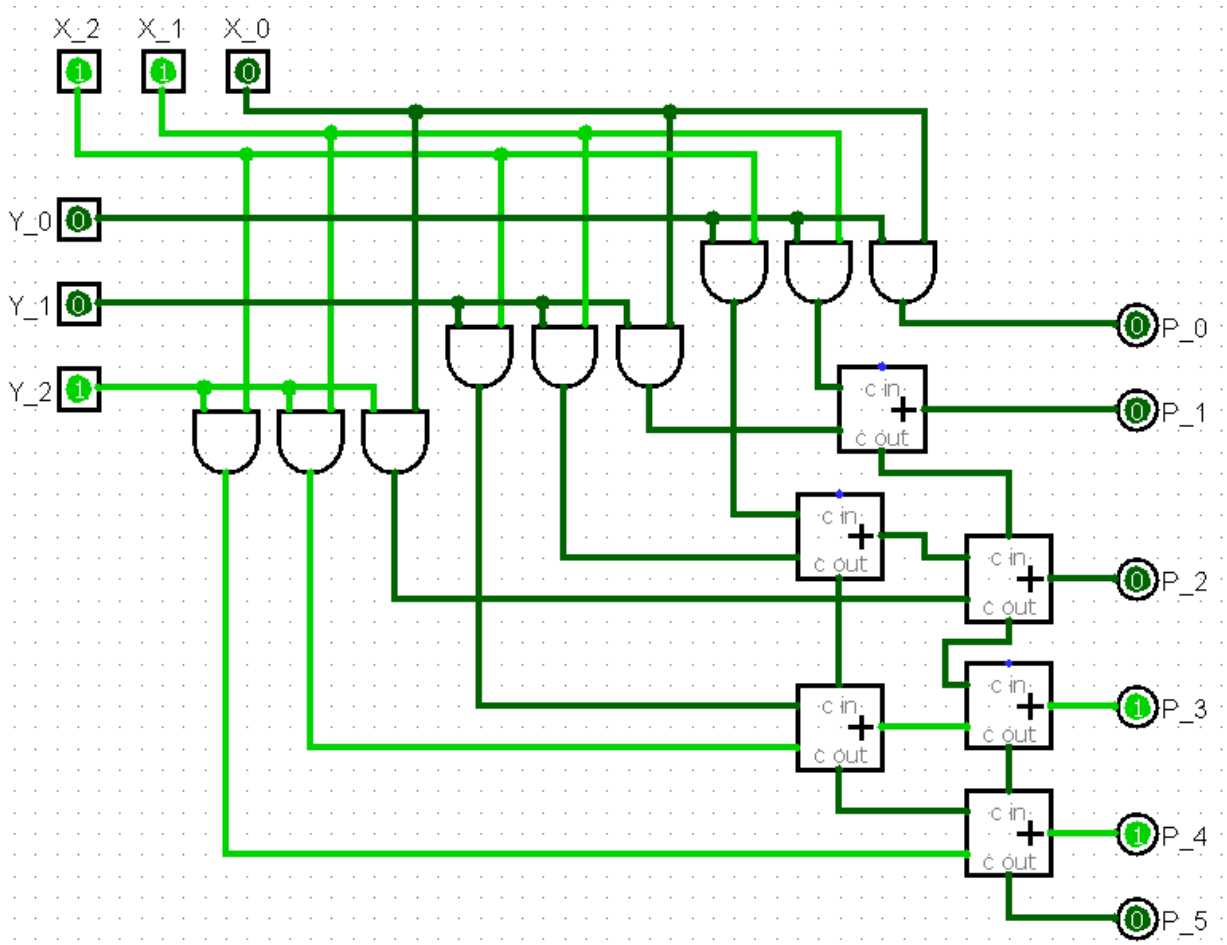
- (Uses 2 half adders)
- (Uses 3 full adders)



Multipliers

3-bit Multiplier Logic Circuit:

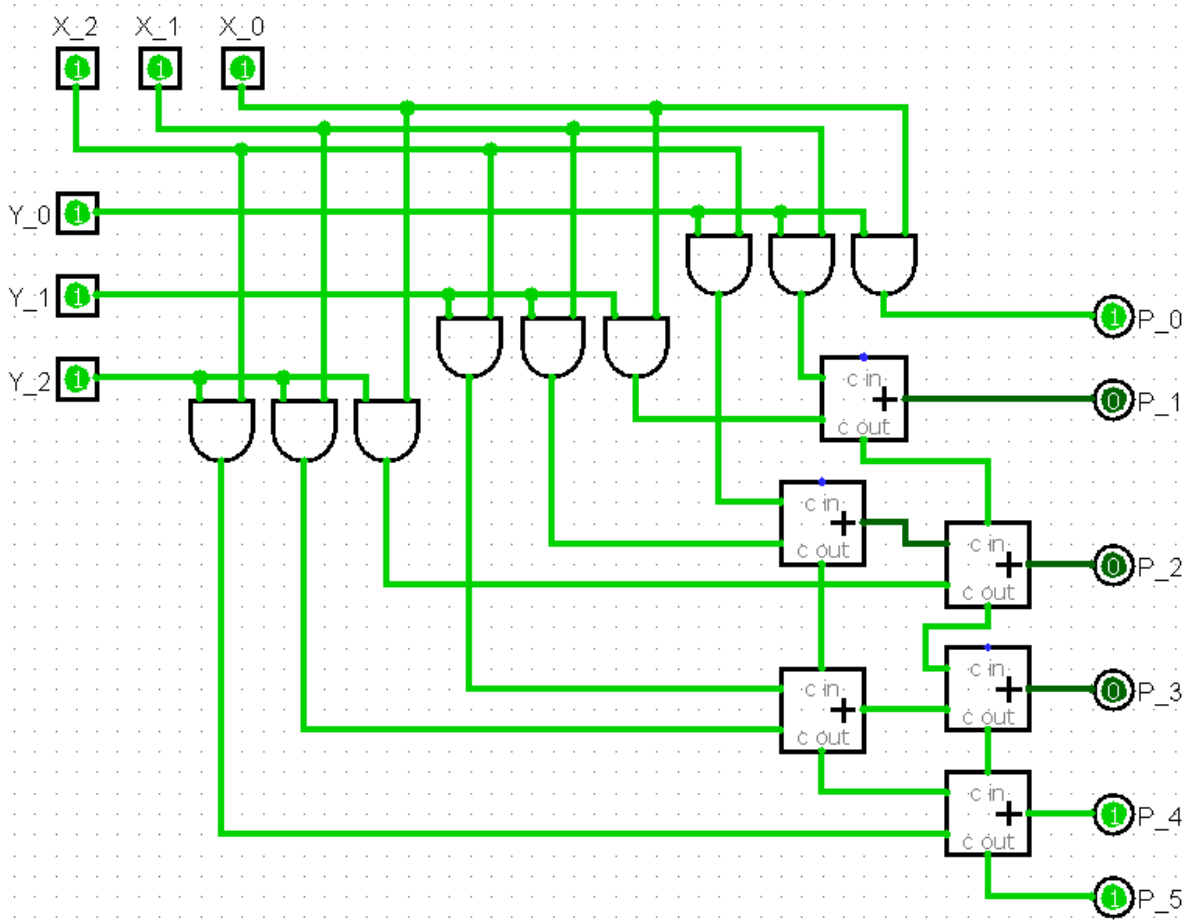
```
      1 1 0
    x 1 0 0
    -------
      0 0 0
    0 0 0
  + 1 1 0
  -------
  0 1 1 0 0 0
```



Multipliers

3-bit Multiplier Logic Circuit:

```
      1 1 1
    x 1 1 1
    -------
      1 1 1
    1 1 1
  1 1 1
+ ---------
  1 1 0 0 0 1
```



Sequential Circuits

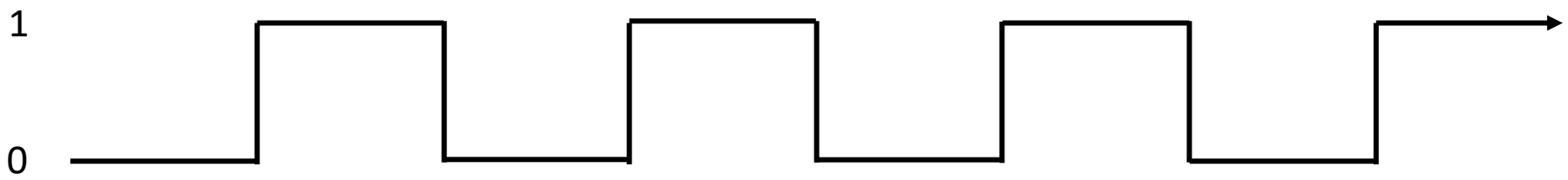
- The combinational logic circuits we've seen have the following limitations:
  - They have no memory capability
  - The output changes as soon as the input changes

- **Sequential logic circuits** maintain a state- The circuit's output is determined by both:
  - The input it currently has
  - The input it has received over time

Clocks

- In most cases, the components in a sequential circuit need to be synchronized
  - The components in the circuit must all update their states at the same time

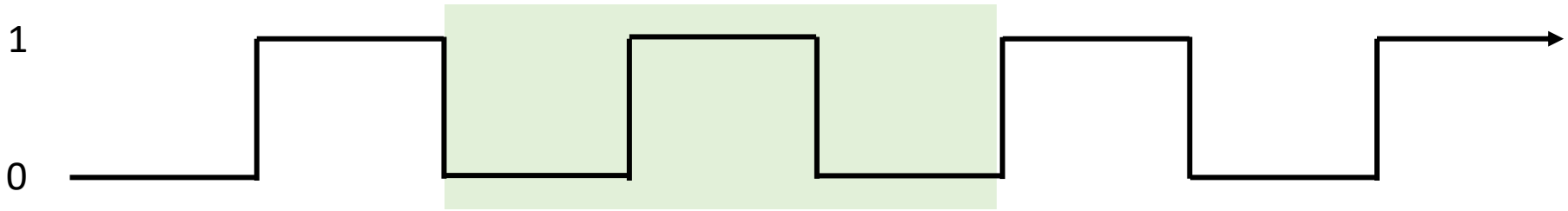
- This is achieved with a **clock signal**
  - A 1-bit signal that alternates between 1 and 0 at regular intervals



Clocks

- A **clock cycle** (“*tick*”) is when the clock transitions from 0 to 1 and back to 0

- A **clock period** is the *time* it takes to complete 1 clock cycle



Clocks

- The **clock frequency** (also called the **clock rate**) is the number of clock periods in some amount of (*wall clock*) time
  - **Wall clock time** is the seconds, minutes, hours, etc. that we experience
  - Time ticks by on a wall clock by the second, whereas these clocks may tick thousands (or millions or billions) of times per one second

- Clock frequency is measured in $\frac{cycle}{second}$ or Hertz (abbreviated Hz)

Clocks

- Clock frequency is the inverse of the clock period (and vice versa)

$$Period = \frac{seconds}{cycle} = \frac{1}{\frac{cycles}{second}} = \frac{1}{Frequency}$$ Seconds per cycle

$$Frequency = \frac{cycles}{second} = \frac{1}{\frac{seconds}{cycle}} = \frac{1}{Period}$$ Cycles per second

Clocks

- Example: What is the clock rate of a processor with a clock period of 250 ps (picoseconds)?
  - $250ps = 250 * 10^{-12}seconds = 0.00000000025s$

- $Period = \frac{250 \times 10^{-12}seconds}{1\ cycle} = \frac{0.00000000025\ seconds}{1\ cycle} = 0.00000000025\ \frac{seconds}{cycle}$

- $Frequency = \frac{1}{Period} = \frac{1}{\frac{250 \times 10^{-12}seconds}{1\ cycle}} = \frac{250 \times 10^{-12}cycles}{1\ second} = \frac{4{,}000{,}000{,}000\ cycles}{1\ second}$

$$= 4{,}000{,}000{,}000\ \frac{cycles}{second} = 4.0\ \text{GHz}$$

Clocks

- Example: What is the clock period of a processor with a clock rate of 3.8GHz?

- $Frequency = 3{,}800{,}000{,}000 \frac{cycles}{second} = \frac{3{,}800{,}000{,}000\ cycles}{1\ second}$

- $Period = \frac{1}{Frequency} = \frac{1}{\frac{3{,}800{,}000{,}000\ cycles}{1\ second}} = \frac{1\ second}{3{,}800{,}000{,}000\ cycles}$

  $= 2.63 \times 10^{-10} \frac{seconds}{cycle} = 263 \times 10^{-12} \frac{seconds}{cycle}$

  $= 263 \frac{picoseconds}{cycle}$

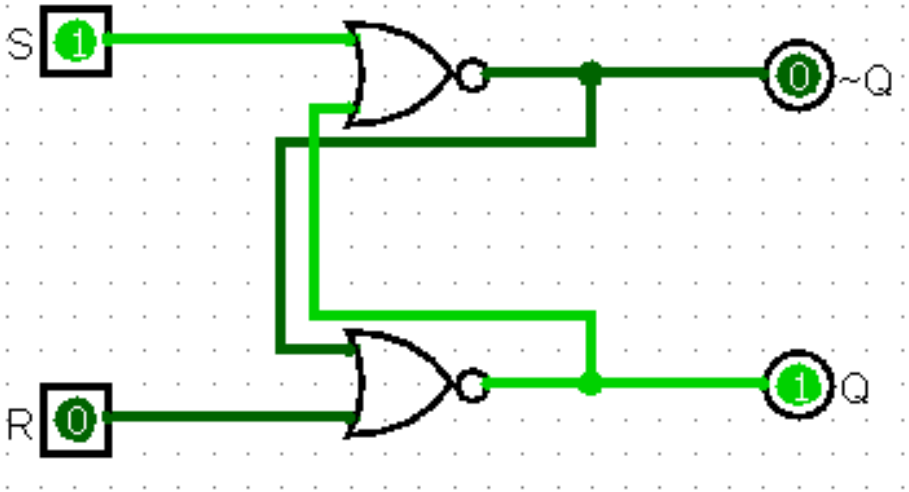
Clocks

- Abstraction of a clock:

SR Latch

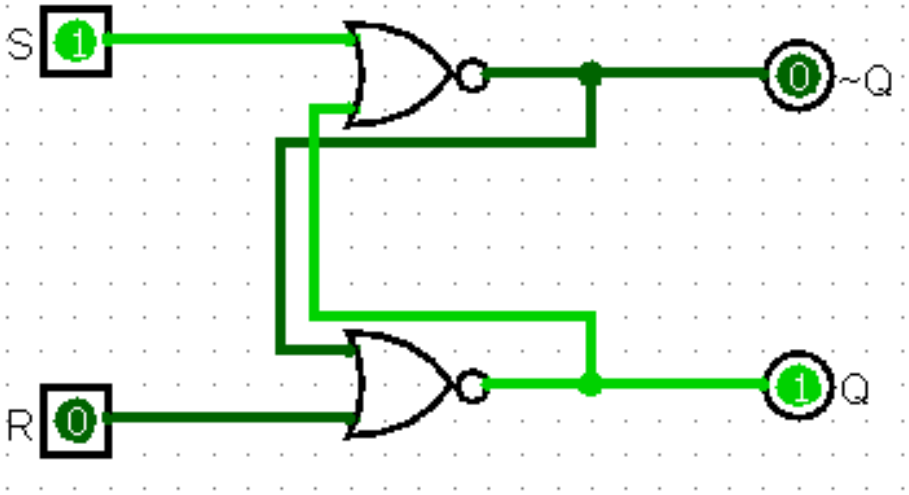
- A **latch** is a digital component that stores 1 bit of information

- An SR Latch has two inputs (Set and Reset) and two outputs (Q and its complement, $\bar{Q}$)



SR Latch

- **Feedback**
  - The output of the top NOR is one input to the bottom NOR
  - The output of the bottom NOR is one input to the top NOR



SR Latch

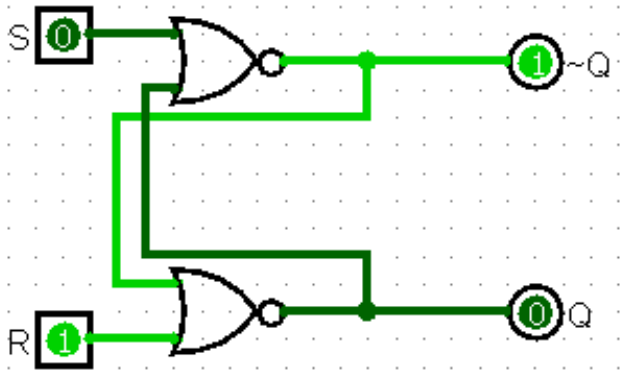
- Recall that, for a NOR gate, the output is 0 if either input is a 1

x	y	$x\ NOR\ y$
0	0	1
0	**1**	**0**
1	**0**	**0**
1	**1**	**0**

SR Latch

- We'll start with the second row of the latch's truth table:
  - This is the Reset state

S	R	$\overline{Q}$	Q	State
0	0			
0	1	1	0	Reset (Q = 0)
1	0			
1	1			

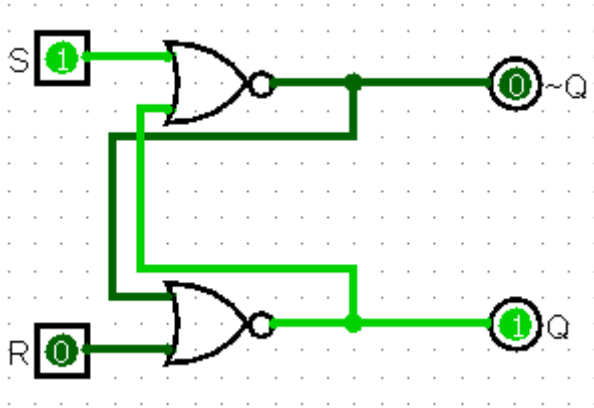


- R is 1, meaning the bottom NOR must have an output (Q) of 0
- S is 0 and the output of the bottom NOR was 0, so the top NOR must have an output ($\overline{Q}$) of 1
  - This is a valid state since Q and $\overline{Q}$ are complements

SR Latch

- Next is the third row of the latch's truth table:
  - This is the Set state

S	R	$\overline{Q}$	Q	State
0	0			
0	1	1	0	Reset (Q = 0)
1	0	0	1	Set (Q = 1)
1	1			

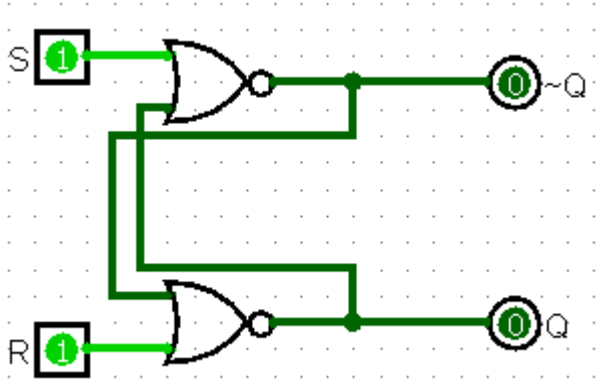


- S is 1, meaning the top NOR must have an output ($\overline{Q}$) of 0
- R is 0, and the output of the top NOR was 0, so the bottom NOR must have an output (Q) of 1
  - This is a valid state since Q and $\overline{Q}$ are complements

SR Latch

- Next is the fourth row of the latch's truth table:
  - This is the Unknown state

S	R	$\bar{Q}$	Q	State
0	0			
0	1	1	0	Reset (Q = 0)
1	0	0	1	Set (Q = 1)
1	1	0	0	Unknown



- S is 1 and R is 1 meaning both NOR gates must have an output of 0
- This doesn't make sense since Q and $\bar{Q}$ are supposed to be complements

SR Latch

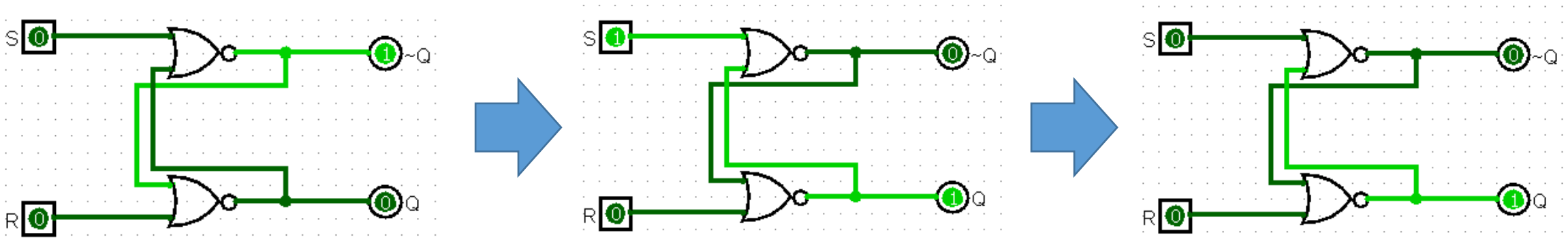
- Back to the first row of the latch's truth table:
  - This is the Unchanged state

S	R	$\bar{Q}$	Q	State
0	0	$\bar{Q}$	Q	Unchanged
0	1	1	0	Reset (Q = 0)
1	0	0	1	Set (Q = 1)
1	1	0	0	Unknown

- S is 0 and R is 0, the current state will depend on the previous state
  - If the previous state was the set state, the current state will still be in the set state
  - If the previous state was the reset state, the current state will still be in the reset state

SR Latch

- Turning Q from 0 (Reset) to 1 (Set)
  - S = 0, S = 1, S = 0

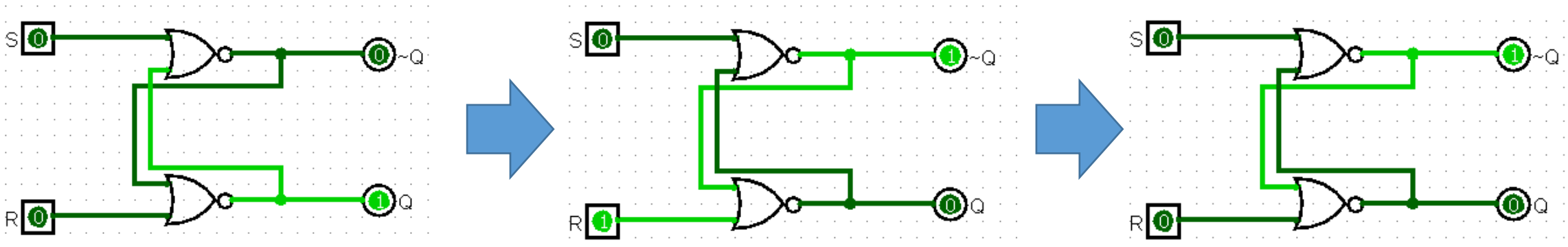


Set

Unchanged

SR Latch

- Turning Q from 1 (Set) to 0 (Reset)
  - R = 0, R = 1, R = 0

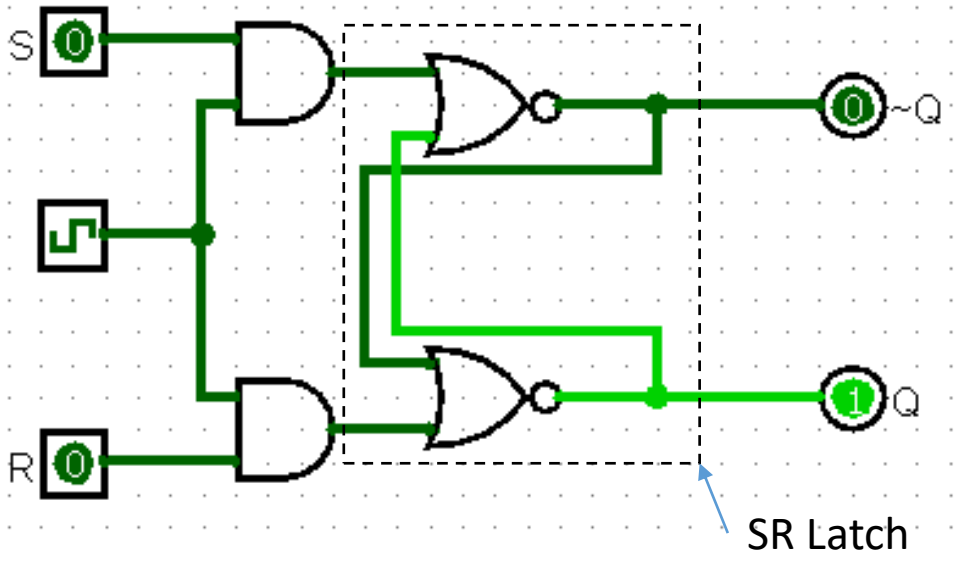


Reset

Unchanged

SR Flip-Flop

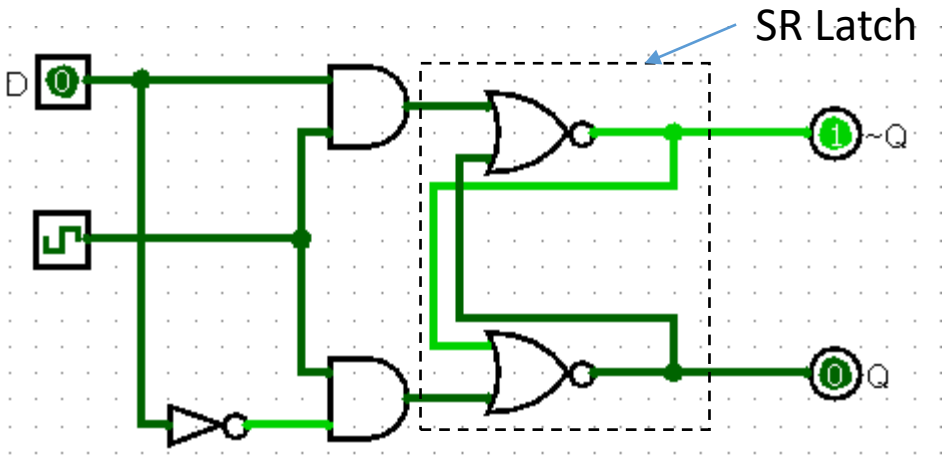
- An extension of the SR Latch is the SR Flip-Flop.
- The S and R inputs are each and'ed with a clock signal.
  - The state can only be changed when the clock signal is 1



S	R	Clock	$\bar{Q}$	Q	*State*
0	0	1	$\bar{Q}$	Q	Unchanged
0	1	1	1	0	Reset (Q = 0)
1	0	1	0	1	Set (Q = 1)
1	1	1	0	0	Unknown
X	X	0	$\bar{Q}$	Q	Unchanged

D Flip-Flop

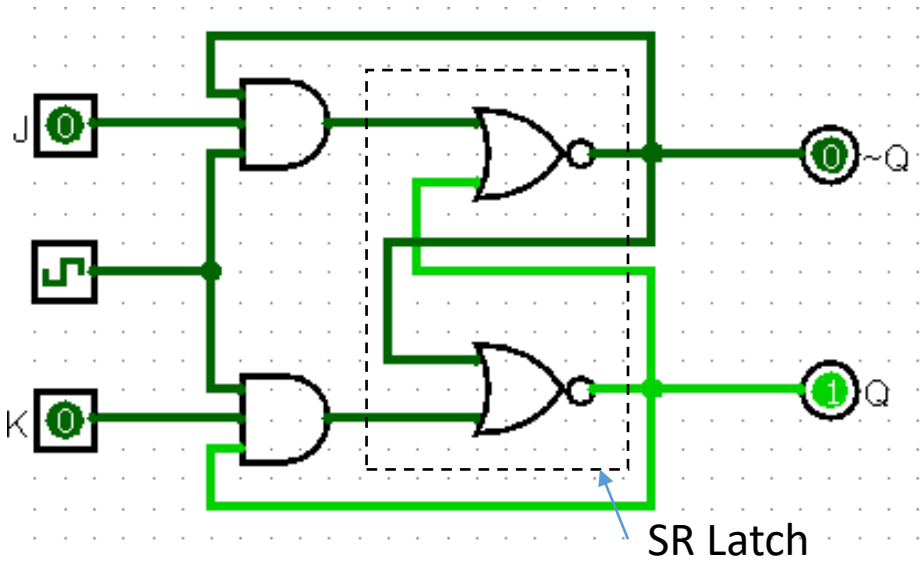
- A D Flip-Flop has one input (D) and two outputs (Q and its complement, $\bar{Q}$)
- The input is and'ed with a clock signal prior to the NOR gates.
  - The state can only be changed when the clock signal is 1



D	Clock	$\bar{Q}$	Q	*State*
0	1	1	0	Reset (Q = 0)
1	1	0	1	Set (Q = 1)
X	0	$\bar{Q}$	Q	Unchanged

JK Flip-Flop

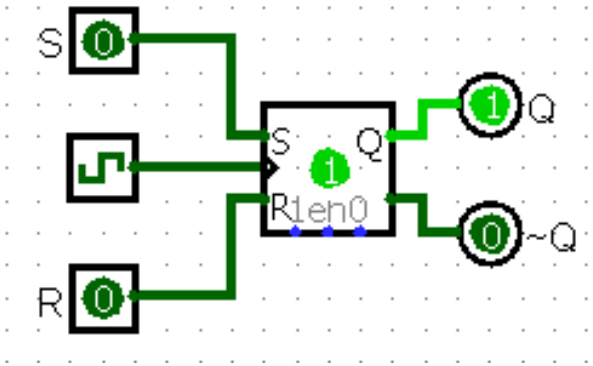
- In a JK Flip-Flop, the input is and'ed with a clock signal *and* an output prior to the NOR gates.
  - The state can only be changed when the clock signal is 1



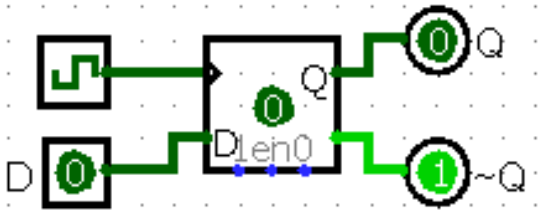
J	K	Clock	$\bar{Q}$	Q	*State*
0	0	1	$\bar{Q}$	Q	Unchanged
0	1	1	1	0	Reset (Q = 0)
1	0	1	0	1	Set (Q = 1)
1	1	1	Q	$\bar{Q}$	Toggle
X	X	0	$\bar{Q}$	Q	Unchanged

Flip-Flops

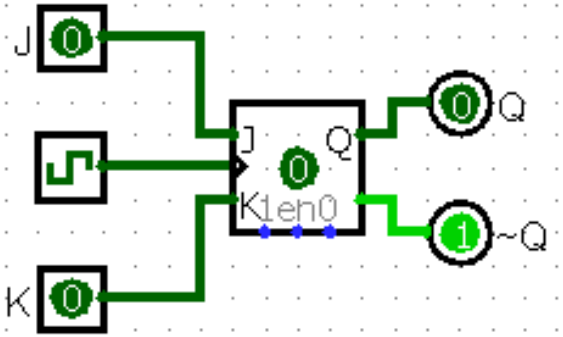
- Abstractions of Flip-Flops:



SR Flip Flop



D Flip Flop



JK Flip Flop

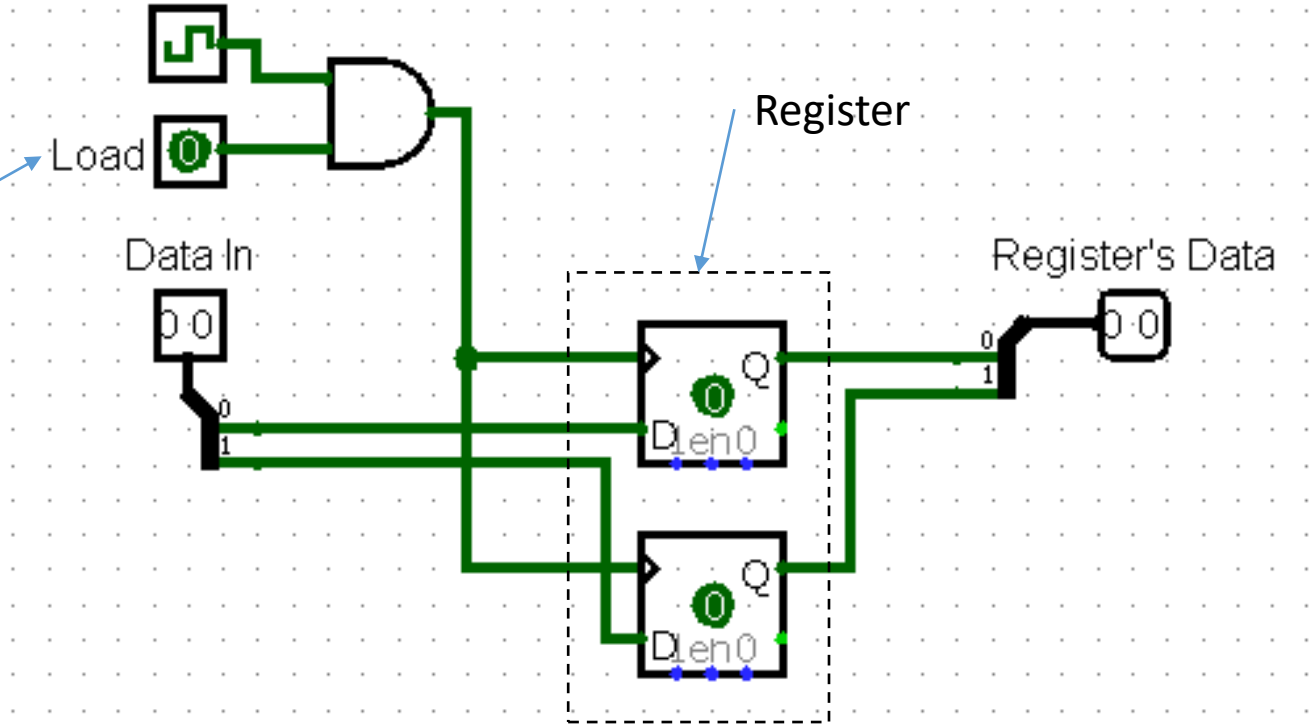
Registers

- By now, we are familiar with the use of registers from assembly programming to temporarily store data.

- Since flip-flops store 1 bit of information, we can create a register from a series of flip-flops
  - 4 flip-flops for a 4-bit register, 32 flip-flops for a 32-bit register, etc.

Registers

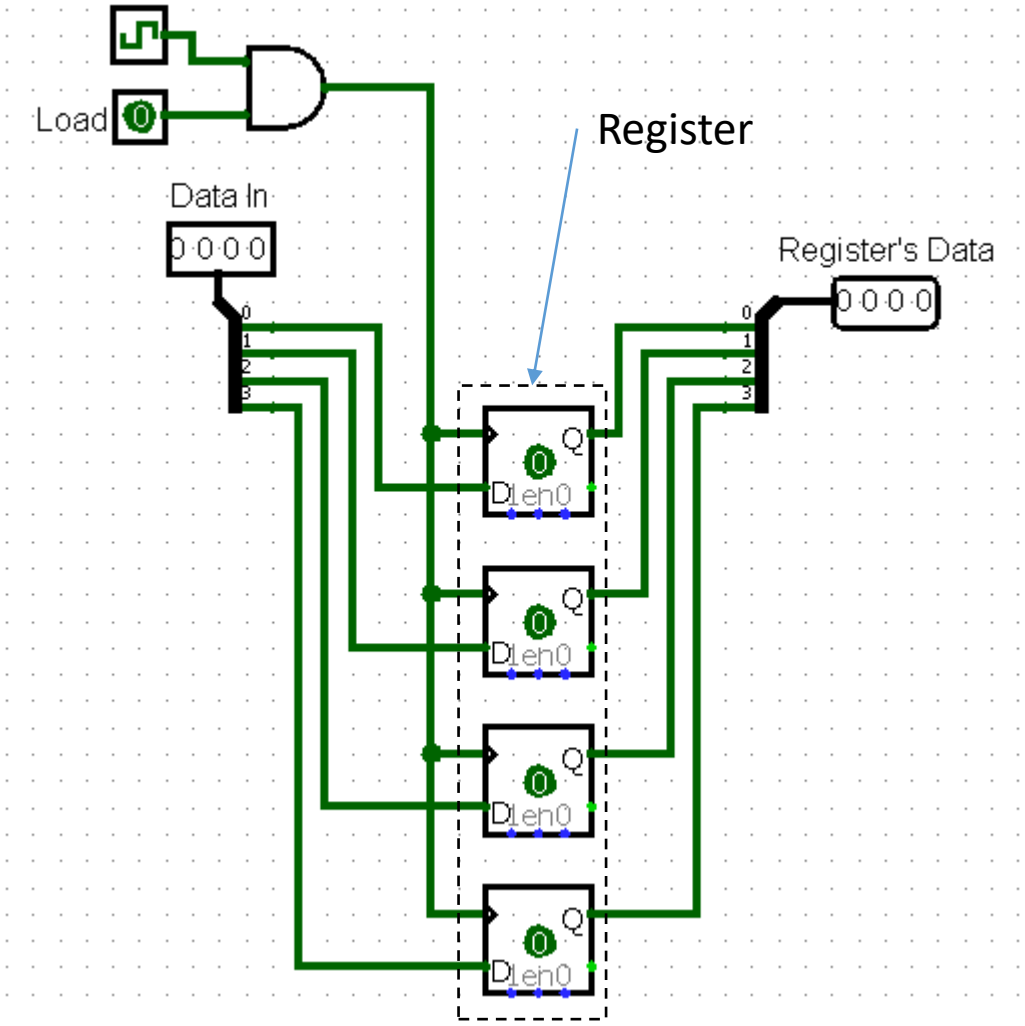
- A 2-bit register:



Enables storage of *Data In* when clock is 1

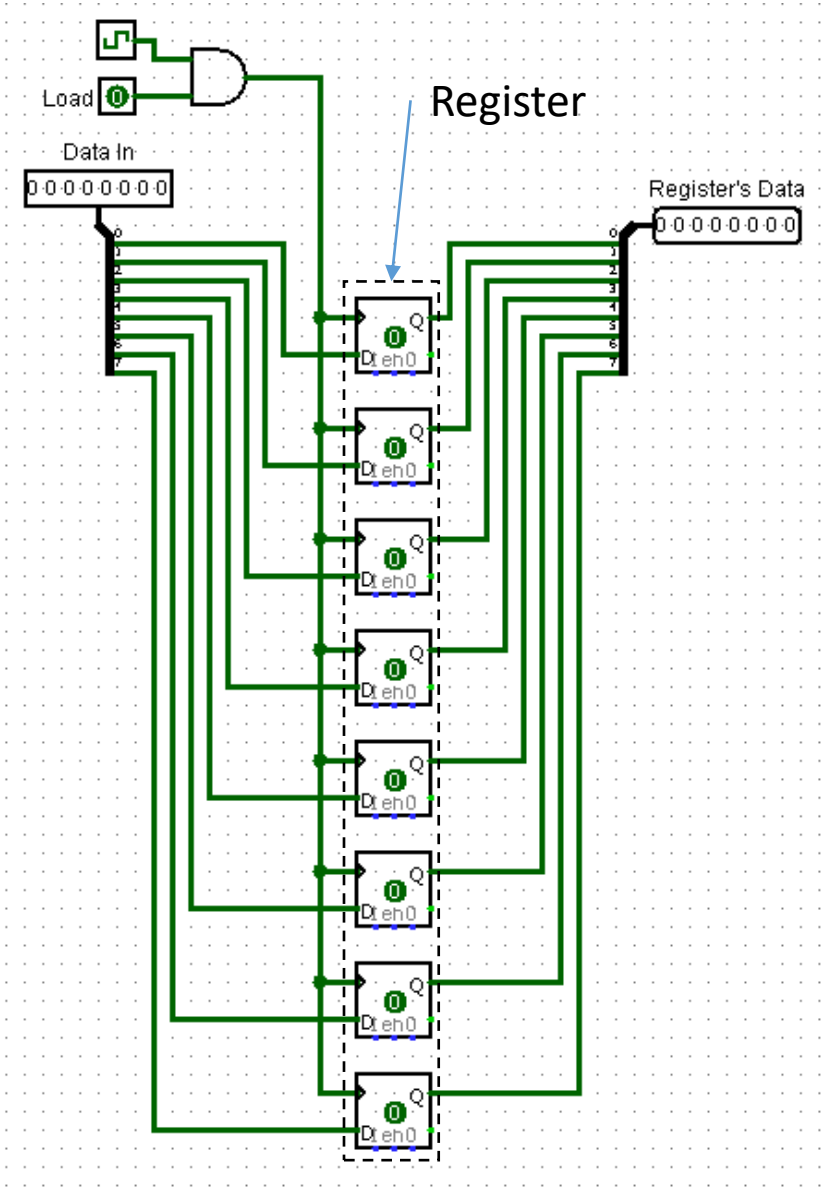
Registers

- A 4-bit register:



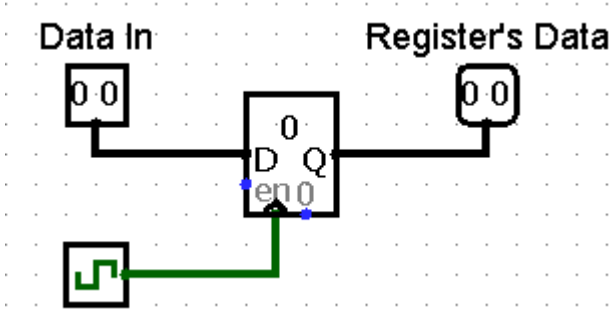
Registers

- An 8-bit register:

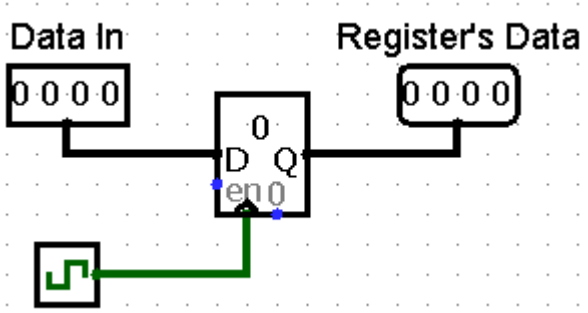


Registers

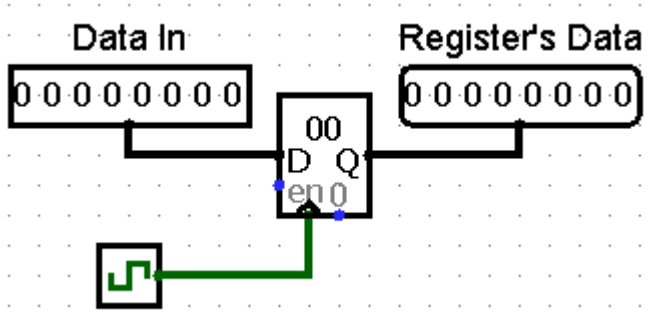
- Abstractions of Registers:



2-Bit Register



4-Bit Register



8-Bit Register