AND

- The (2-input) AND operator takes 2 boolean inputs, and produces a boolean output.
- C = A & B

		B	
	AND	False	True
A	False	False	False
	True	False	True

Truth Table

		B	
	&	0	1
A	0	0	0
	1	0	1

Binary Truth Table

0: False

1: True

AND (contd.)

True & False = False

False & False = False

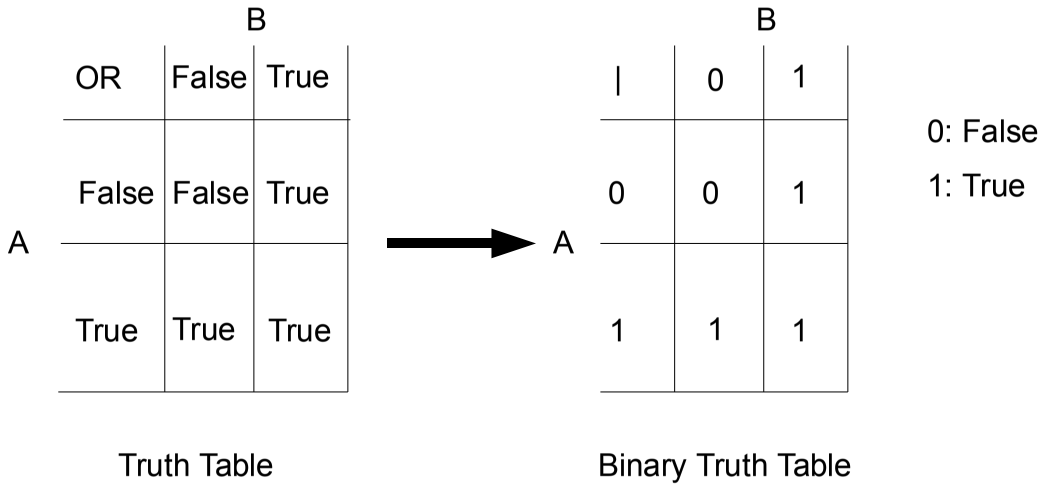
True & True = True

1 & 0 = 0

1 & 1 = 1

OR

- The (2-input) OR operator takes 2 boolean inputs, and produces a boolean output.
- C = A | B



OR (contd.)

True | False = True

False | False = False

True | True = True

1 | 0 = 1

1 | 1 = 1

0 | 0 = 0

NOT

- The unary NOT operator takes a boolean input, and produces a boolean output.
- B = ¬A

A	NOT A
False	True
True	False

Truth Table

A	¬A
0	1
1	0

Binary Truth Table

XOR

- The (2-input) XOR operator takes 2 boolean inputs, and produces a boolean output.
- C = A ⊗ B

	XOR	B: False	B: True
A	False	False	True
A	True	True	False

Truth Table

	⊗	B: 0	B: 1
A	0	0	1
A	1	1	0

Binary Truth Table

0: False

1: True

"detects differences"

XOR (contd.)

True ⊗ False = True

False ⊗ False = False

True ⊗ True = False

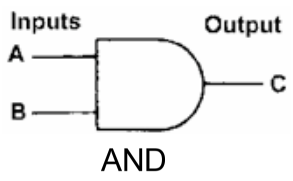
1 ⊗ 0 = 1

1 ⊗ 1 = 0

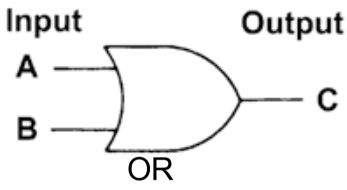
0 ⊗ 0 = 0

Gates

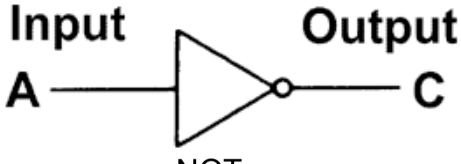
- In addition to representing AND, OR, NOT and XOR by &, |, ¬ and ⊗, these operators can be represented by the following gate symbols.



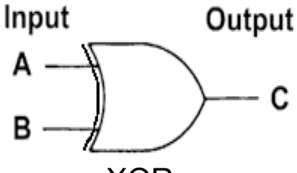
AND



OR



NOT



XOR

Properties of Boolean Logic

- Commutativity
  - A & B = B & A
  - A | B = B | A
- Associativity
  - A & (B & C) = (A & B) & C
  - A | (B | C) = (A | B) | C
- Absorption
  - A & (A | B) = A
  - A | (A & B) = A
- Distributivity
  - A & (B | C) = (A & B) | (A & C)
  - A | (B & C) = (A | B) & (A | C)
- Complements
  - A & ¬A = 0
  - A | ¬A = 1
- Identity rules
  - A & 1 = A
  - A | 0 = A
- Boundedness
  - A & 0 = 0
  - A | 1 = 1
- Involution
  - ¬¬A = A

Manipulating boolean expressions

- Simplify X & (Y | ¬X)
  - (X & Y) | (X & ¬X) - distributivity
  - (X & Y) | (0) - complements
  - X & Y - identity
- Prove A | A = A
  - You must use the properties of boolean logic.
  - A | A
  - A | (A &1) - identity
  - A - absorption

Manipulating binary expressions (2)

- Prove (A & B) | (A & ¬B) | (¬A & B) = A | B
  - (A & B) | (A & ¬B) | (¬A & B)
  - (A & (B | ¬B)) | (¬A & B) - distributivity
  - (A & 1) | (¬A & B) - complements
  - A | (¬A & B) - identity
  - A | ((¬A & B) | 0) - identity
  - A | ((¬A & B) | (A & ¬A)) - complements
  - A | (¬A & (B | A)) - distributivity
  - (A | ¬A) & (A | (B | A)) - distributivity
  - 1 & (A | (B | A)) - complements
  - (A | A) | B - identity, commutativity & associativity
  - A | B - see proof on previous slide

Manipulating binary expressions (3)

- Simplify X | (¬X & Y)
  - (X & 1) | (¬X & Y) - identity
  - (X & (1 | Y)) | (¬X & Y) - identity
  - ((X & 1) | (X & Y)) | (¬X & Y) - distributivity
  - (X | (X & Y)) | (¬X & Y) - identity
  - X | ((X & Y) | (¬X & Y)) - associativity
  - X | (X | ¬X) & Y) - distributivity
  - X | (1 & Y) - complements
  - X | Y - identity

DeMorgan's theorems

- From the properties of boolean algebra we can derive 2 very useful theorems.
  - ¬(A & B) = ¬A | ¬B
  - ¬(A | B) = ¬A & ¬B
- These are called DeMorgan's theorems, named after the mathematician who discovered them.
- Simplify ¬(A | ¬(B & C))
  - ¬(A | (¬B | ¬C)) - DeMorgan
  - ¬A & ¬(¬B | ¬C) - DeMorgan
  - ¬A & ¬¬B & ¬¬C - DeMorgan
  - ¬A & B & C - involution

DeMorgan's theorems (2)

- Simplify ¬((¬A & B) | (¬A & ¬B) | ¬B)
  - ¬((¬A & (B | ¬B)) | ¬B) - distributivity
  - ¬((¬A & 1) | ¬B) - complements
  - ¬(¬A | ¬B) - identity
  - ¬(¬(A & B)) - de Morgan
  - A & B - involution

Karnaugh Maps

- We can use the properties (axioms) of boolean algebra to simplify boolean expressions.
- There is also a graphical technique, ***Karnaugh Maps***.
  - Split the boolean variables into 2 groups (as even as possible).
  - Arrange as a rectangular array.

CD \ AB	00	01	11	10
00				
01				
11				
10				

The numbering on the axes is critical. It is called Gray coding. Between any successive codes only one digit is changed.

C \ AB	00	01	11	10
0				
1				

Karnaugh Maps (2)

- Let us simplify
  - ((W | ¬X | Z) & Y & (W & Y)) | (W & X & Y & Z)
- First fill in the truth table for this expression
  - Calculate the value of the expression (R) for each value of W, X, Y, Z

W	X	Y	Z	R
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0

W	X	Y	Z	R
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Karnaugh Maps (3)

- Copy the values from the truth table to the Karnaugh Map.
  - Warning!!! The Karnaugh Map uses Gray coding so be careful translating the results from the truth table to the Karnaugh Map.

YZ \ WX	00	01	11	10
00				
01				
11			1	1
10				1

Assume empty cells are 0.

Karnaugh Maps (4)

- Group adjacent cells with '1' together. The size of each group must be a power of 2 (2,4,8,...).
  - Make the groups as big as possible.

YZ \ WX	00	01	11	10
00				
01				
11			1	1
10				1

In our case we can get 2 groups of 2.

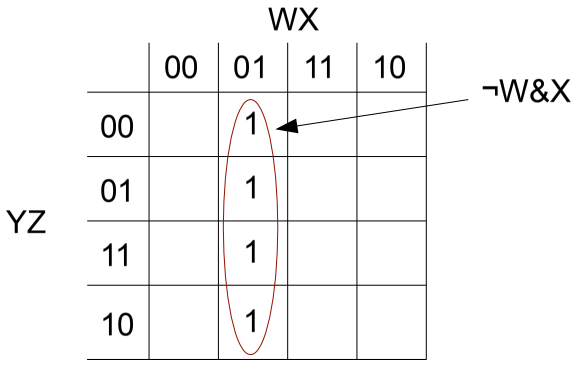
In this group W,Y and Z do not change values. X has the values 1 and 0, and therefore is not needed.
This group represents the term W&Y&Z.

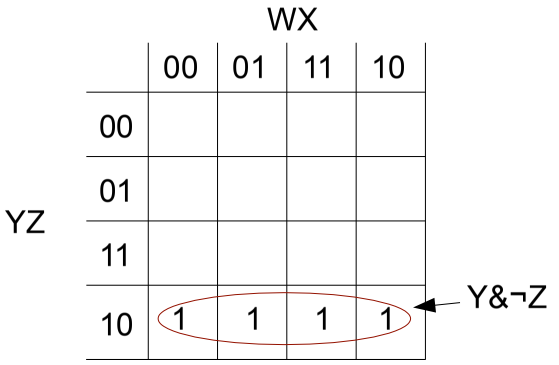
In this group W,X and Y do not change values. Z has the values 1 and 0, and therefore is not needed.
This group represents the term W&¬X&Y.

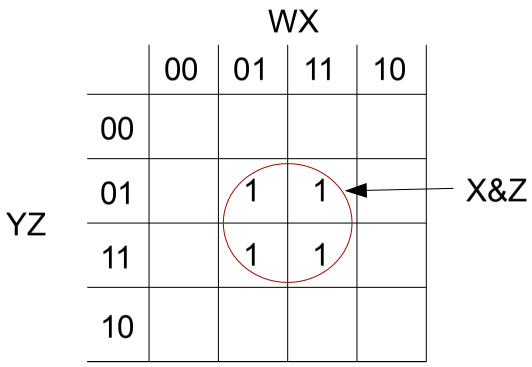
The complete map represents all the groups 'OR'ed together.
(W&Y&Z) | (W&¬X&Y)
=(W&Y)&(Z|¬X)

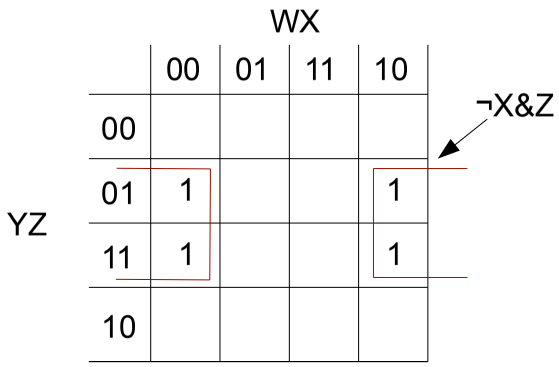
Karnaugh Maps (5)

- Some examples of groups of 4.



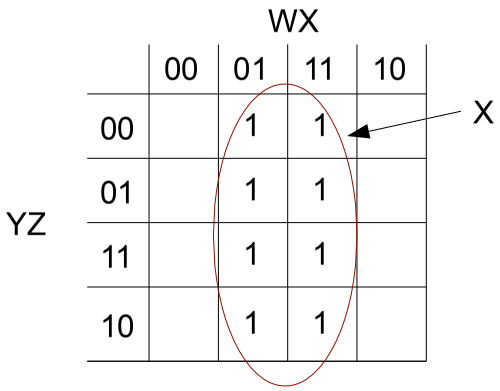


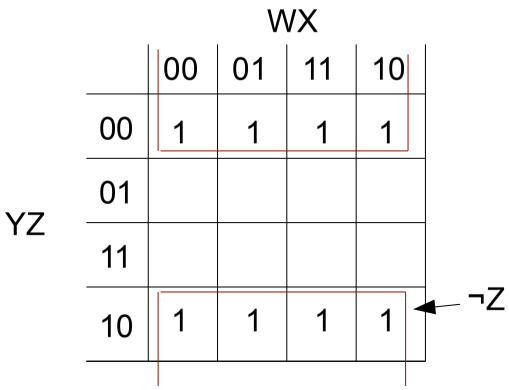




Karnaugh Maps (6)

- Some examples of groups of 8.





Karnaugh Maps (7)

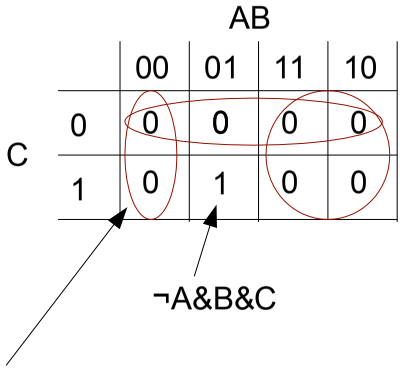
- Grouping '1's together generates a ***sum of products*** expression.
  - AND can also be represented as '.'
  - OR can also be represented as '+'
  - (A&B) | (A&¬C) | (B&C) can be written as A.B + A.¬C + B.C
- Grouping '0's together generates a ***product of sums*** expression.
  - (A+¬B+C).(¬A+¬C).(B+C)
  - which is also (A|¬B|C)&(¬A|¬C)&(B|C)

Karnaugh Maps (8)

- Simplify ¬(A | ¬(B & C)) using Karnaugh Maps.

A	B	C	R
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

truth table

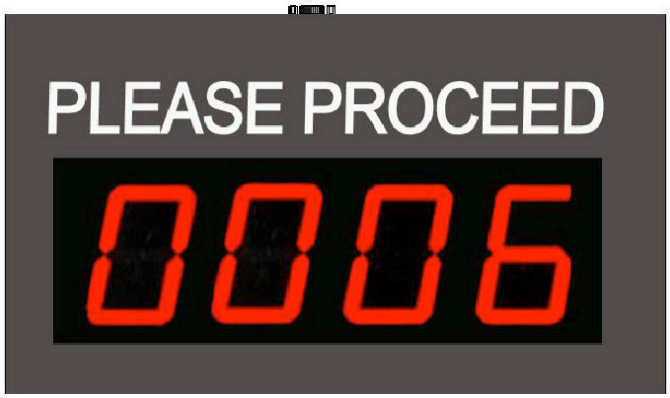


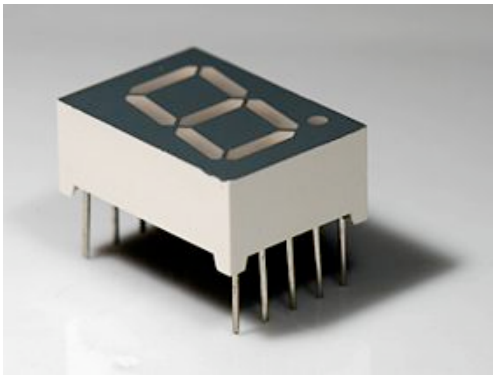
Or, grouping the '0's together:
(¬A|¬B)&(¬C)&(A)

As an exercise show
¬A&B&C = (¬A|¬B)&(¬C)&(A)
using boolean algebra.

Example Application: 7 segment LED display

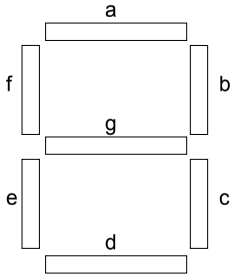
- A common way on displaying data on small embedded computers is a 7 segment display.
  - Computers are everywhere!!! There not just desktops and laptops. They are phone, microwave ovens, house alarms, cars (from 2 to 15+ in high-end cars).



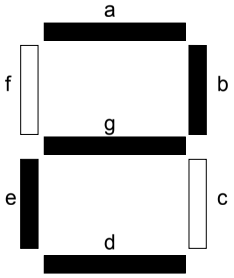


7 segment display (2)

- Each display is made up of 7 individual light emitting diodes (LEDs). By combining several of these LEDs you can form a character.



The number 2 would have the 'a', 'b', 'g', 'e' and 'd' segments turned on.

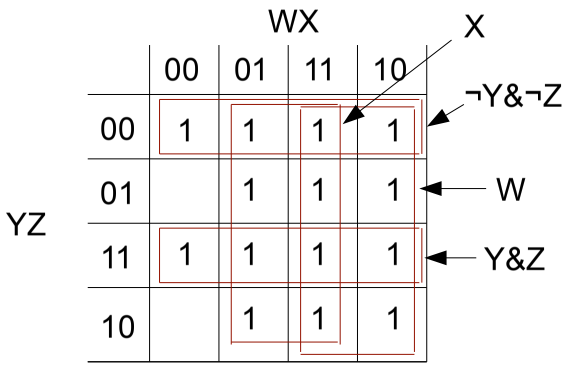


7 segment display (3)

- The task is to convert a decimal number (coded as binary) into a pattern of 7 signals to the display, one for each LED.

	BCD										
	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	1	1	0	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1

Let's consider the 'c' segment.



W | X | (¬Y&¬Z) | (Y&Z)

7 segment display (4)

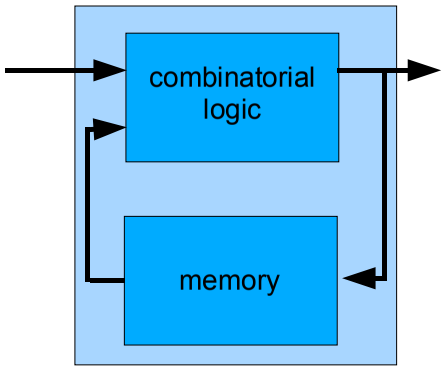
- You can repeat this for the other 6 segment (a,b,d,e,f and g)
  - Do this as an exercise.
- Remember that you can group '0's instead of '1's
  - **but** the result is a product of sums expression and not a sum of products expression.
-
- Gray coding for 3 variables is:
  - 000 001 011 010 110 111 101 100

Sequential Logic

- The type of logic that we have been looking at is called combinatorial logic.
  - The value of the output only depends on the current value of the inputs. Previous results have no effect on the current result.
- In sequential logic the current output not only depends on the current inputs but also on previous outputs.
  - There is memory.
  - Sequential logic is a vital element of the architecture of a computer. Without it there would be no memory; a processor would not be able to store results or use previous results in calculations or decisions.

Sequential Logic (2)

- Combinatorial logic is built from boolean operators (or gates in electronic terms).
- Sequential logic is built from ***latches***.
  - Also called bistables or flip-flops.
- We can think of sequential logic as combinatorial logic combined with memory.
  - The current inputs are combined with a memory of the previous output to produce the next output, which is then stored in the memory.
  - current inputs + memory → ***current state***
  - The transition from the current state to the ***next state*** is usually controlled by a clock signal.

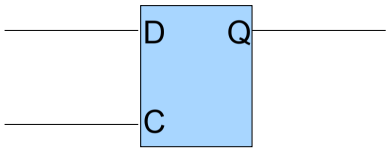


Sequential Logic (3)

- The major applications of sequential logic are:
  - **Latch**: A 1-bit memory
  - **Register**: This is a series of *m* latches that can store an *m*-bit word.
  - **Shift Register**: A special purpose register that can shift its contents to the left or right. If an 8-bit shift register initially contains 00101010 and is shifted 1 bit to the left, it will then contain 01010100.
    - Evaluate the 2 binary numbers above and see if you can determine the effect of a 1 bit shift left. You will see why shift registers are important.
  - **Counter**: A special purpose register that increases its contents by 1 every time it is clocked.

D flip-flop

- The D flip-flop has 2 inputs, D and C and 1 output, Q.
- The D input is the *data input* and the C input in the *clock input*.
- Whenever C is 0, Q retains its value.
- Whenever C is 1, Q takes the value of the D input.



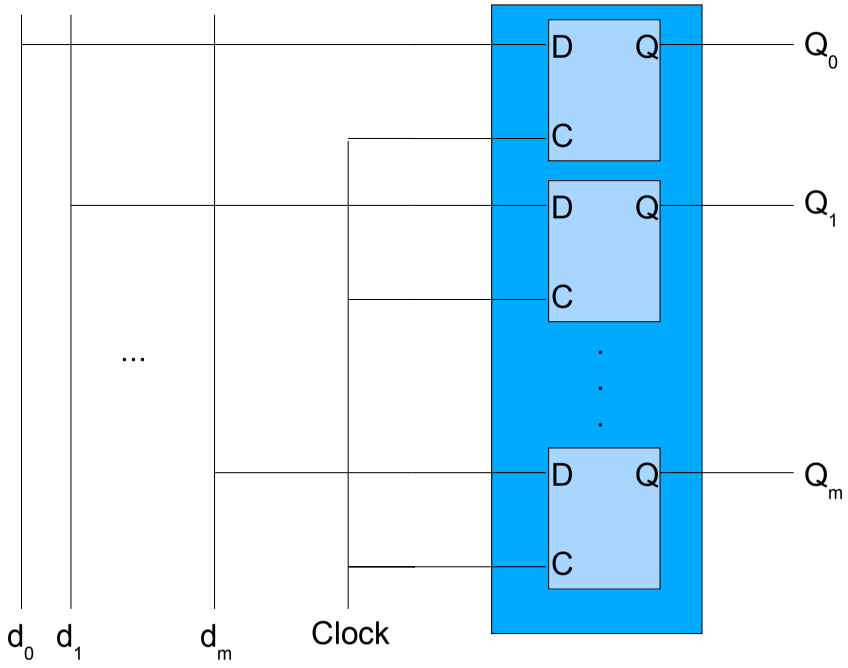
D	C	Q^+
0	0	Q
1	0	Q
0	1	0
1	1	1

Q: current state
Q^+: next state

Generally the clock is *level* sensitive in D flip-flops.

D flip-flop (2)

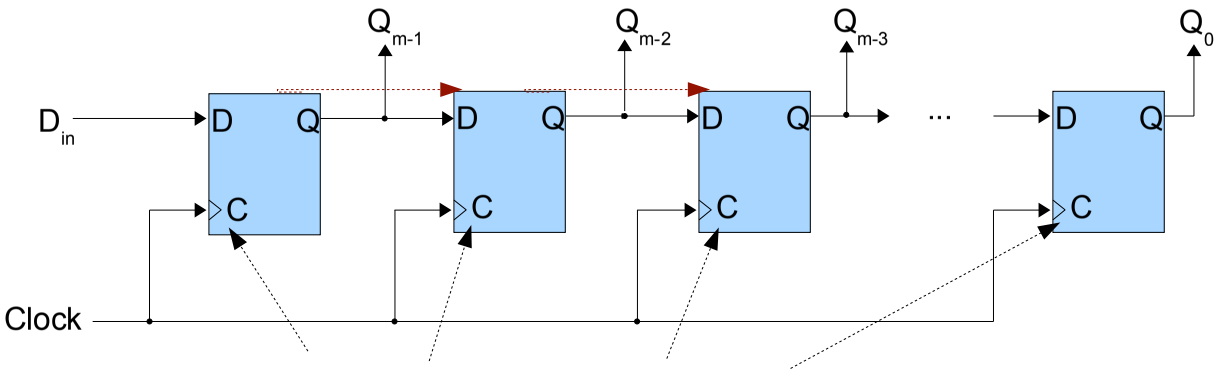
- D flip-flops can be combined to create a register.



D flip-flop (3)

- D flip-flops can be combined to create a right-shift register.

At each clock pulse data is copied from one D flip-flop to the D flip-flop to its right.

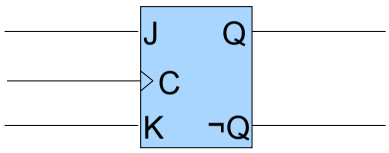


These D flip-flops are edge-triggered. The rising clock edge triggers the transition to the next state.

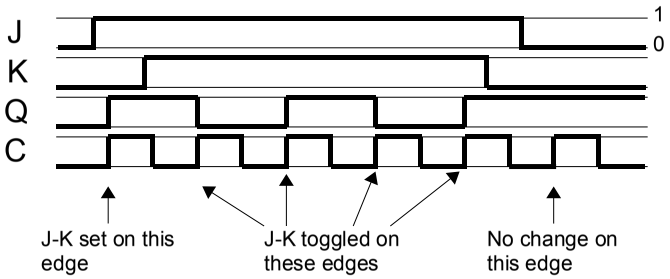
What would happen if they were level-sensitive clocks?

J-K flip-flop

- The J-K flip-flop has 3 inputs, J, K and C and 2 output, Q and ¬Q.
- The C input is the *clock input* and it is edge triggered.
- The ¬Q output is the negation of the Q output.

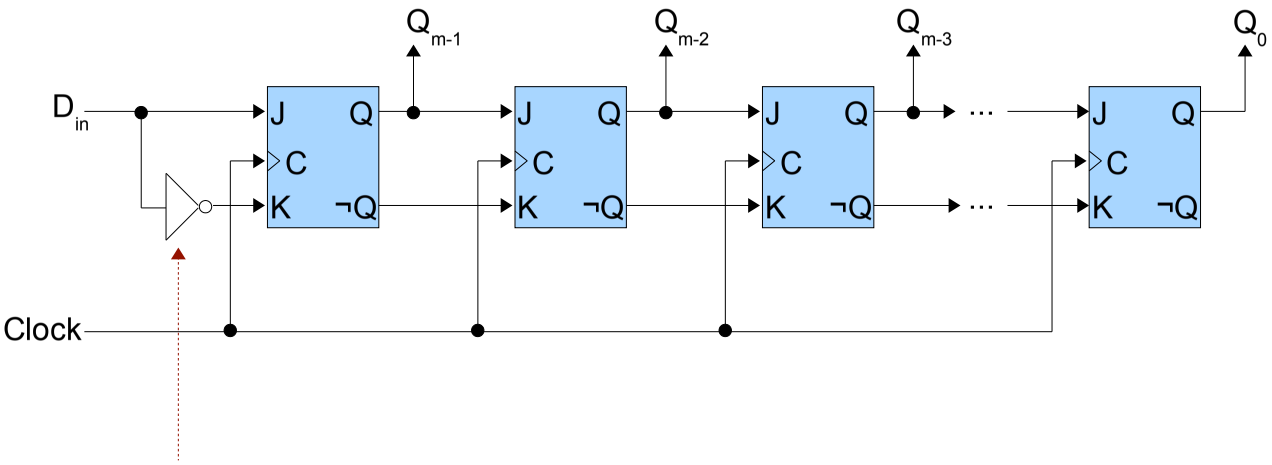


J	K	Q^+	
0	0	Q	No change
0	1	0	Clear
1	0	1	Set
1	1	¬Q	Toggle



J-K flip-flop (2)

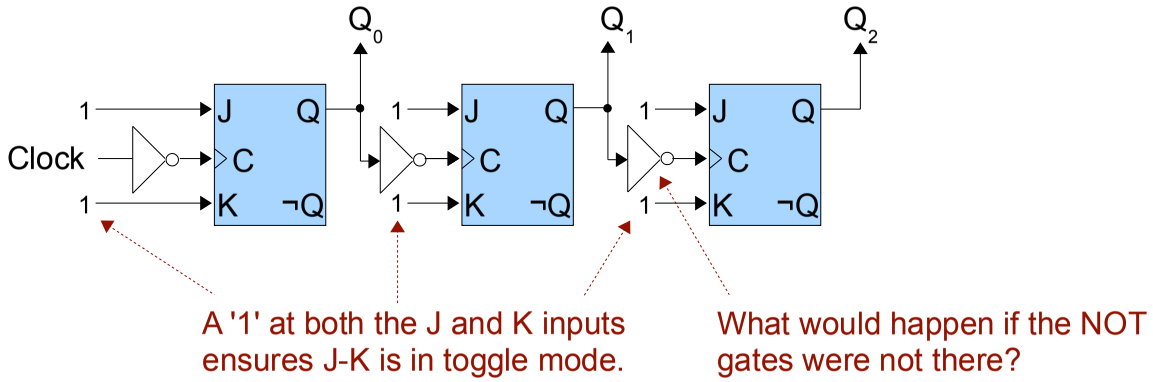
- J-K flip-flops can be used to build a shift register.



Inverter (NOT gate) ensures J-K is in set mode when D_{in} is 1, and in clear mode when D_{in} is 0.

J-K flip-flop (3)

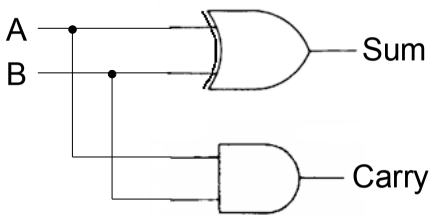
- J-K flip-flops can be used to build binary counter.
- For a 3-bit up counter the sequence of values will be:
  - 000, 001, 010, 011, 100, 101, 110, 111
  - The least significant bit (LSB) changes state every clock.
  - Bit *i* changes state whenever bit *(i-1)* goes from 1 to 0.



Half Adder

- When adding 2 1-bit numbers, the following are the possibilities.
  - 0 + 0 = 0 0 + 1 = 1 1 + 0 = 1
  - 1+1 = 0 and there is a carry forward.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Sum = A ⊗ B
Carry = A & B

Full Adder

- A full adder includes the possibility of a carry coming in from a previous calculation.

A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

As an exercise work out the boolean expressions for Sum and Carry Out.