COSC2473

Digital Logic

Logic Gates & Bit Masking

Boolean Algebra

Digital Logic

- Binary can be considered as truth values:

  0 = False, 1 = True

- Boolean Algebra: the laws of algebra for truth values
  - consists of operations on truth values, the result of which is another truth value
  - fundamental for the design of logic circuits and for tests in computer programming

- CPU hardware consists of circuits built from logic gates
  - logic gates perform Boolean Algebra

Boolean Operators in Python

- Python does not have a Boolean data type directly.
  - There is variable that you can declare as Boolean, but there is a Boolean value type, and a function bool() that returns it.
- A Boolean value type is True or False (not true or TRUE)
  - Any value can be converted to Boolean
    - Bool(“hello”)=True, bool(3)=True, bool(0)=False, bool(“”)=False $\overline{x}$
- In Java and Python, Boolean operations come in:
  - Real Boolean operators with True/False values
    - These are constructions within the language (e.g. Objects)
  - Bitwise Boolean operators
    - these operate on the bits of (usually) integers
- The bitwise operators are generally more useful.

<u>We will show only bitwise operators from now on.</u>

Bitwise operators

- The bitwise operators work the same in Java and Python

Notation	Operation
~a	1's complement of a
a & b	AND
a | b	OR
a ^ b	XOR
a << n	Left shift by n bits
a >> n	Right shift by n bits

- aa

NOT Operator

- Boolean Operator: NOT
  - NOT returns the opposite value
    - i.e. True becomes False, False becomes True

NOT	-
0	1
1	0

(written as /x or -x or sometimes !x or $\overline{x}$)

Java:

```
int x, a=1;          // a = 0x01, 0000 0001₂
x = ~a;              // x = 0xFE, 1111 1110₂ = 254
```

Python

```
 a = 0
~a = -1      # since '11111…11' = -1
```

AND Operator

- AND is true when ALL inputs are true
  - written as: $x.y$ or $x \wedge y$ or xy

AND	.
0 . 0	0
0 . 1	0
1 . 0	0
1 . 1	1

Java:

```
int x, a = 0x18, b = 0x11;
x = a & b;        // a = 0001 1000₂
                  // b = 0001 0001₂
                  // x = 0001 0000₂ = 0x10 = 16
```

OR Operator

- OR is true when ANY input is true
  - written as: $x+y$ or $x \vee y$

OR	+
0 + 0	0
0 + 1	1
1 + 0	1
1 + 1	1

Java:

```
int x, a = 0x18, b = 0x11;
x = a | b;          // a = 0001 1000₂
                    // b = 0001 0001₂
                    // x = 0001 1001₂ = 0x19 = 25
```

XOR Operator

- XOR (exclusive OR) is similar to OR, except it is true when only one, not both, input is true
  - written as $x \oplus y$

XOR	$\oplus$
0 . 0	0
0 . 1	1
1 . 0	1
1 . 1	0

Java:

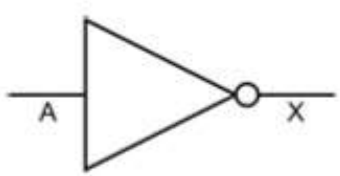
```
int x, a = 0x18, b = 0x11;
 x = a ^ b;        // a = 0001 1000₂
                   // b = 0001 0001₂
                   // x = 0000 1001₂ = 0x09 = 9
```

Logic NOT gate

- *Computer hardware is made from logic gates*
  - *Basic logic gates: AND, OR, NOT*
  - *Derived negated gates: NAND, NOR, XOR, XNOR*

- *Any logic circuits can be build from combinations of just AND, OR, NOT*
  - *alternatively, use just NAND, NOR*

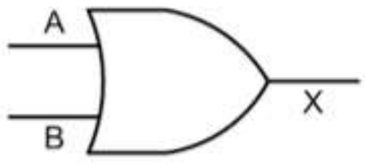
- *NOT gate*



  - *The above triangle is actually the symbol for an "Op Amp", an operational amplifier whose function doubles to clean up the digital signal as it traverses the circuit. The circle to the left is the thing that denotes the actual NOT operation*

Logic OR / AND gates

- OR gate

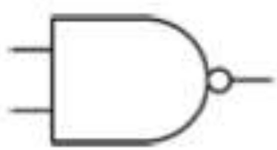


- AND gate

Logic XOR / NAND gates

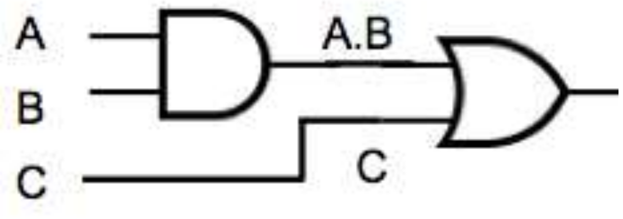
- XOR gate

- NAND gate (a combination of NOT AND)
  - Note the little circle to the right.

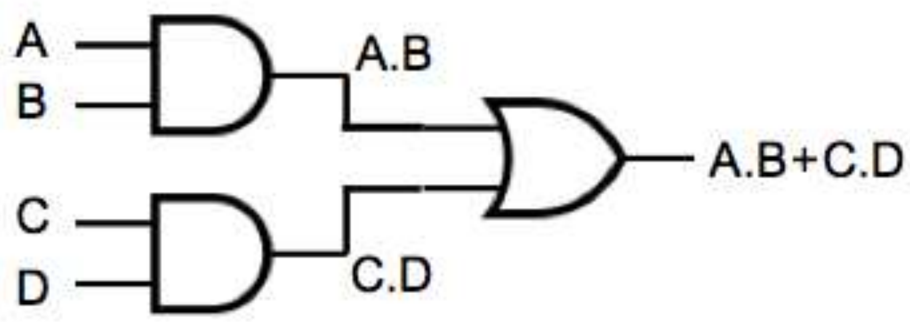


Logic Circuits

- A.B+C



- A.B+C.D



Gating

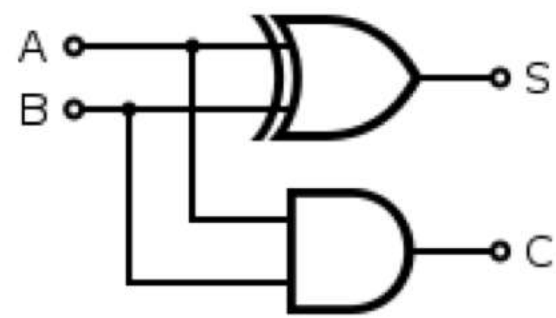
- Normally, AND, OR, XOR have pure signal inputs, but what if one of the signals is a constant?
- AND
  - 0 AND A = 0, always
  - 1 AND A = A, always
- OR
  - 0 OR A = A, always
  - 1 OR A = 1, always
- XOR
  - 0 XOR A = A
  - 1 XOR A = ~A

Half Adder

- Say we wanted to construct an adder, which adds two bits together and outputs the sum (as a single bit) and a carry bit
  - called a *half adder*
  - truth table:

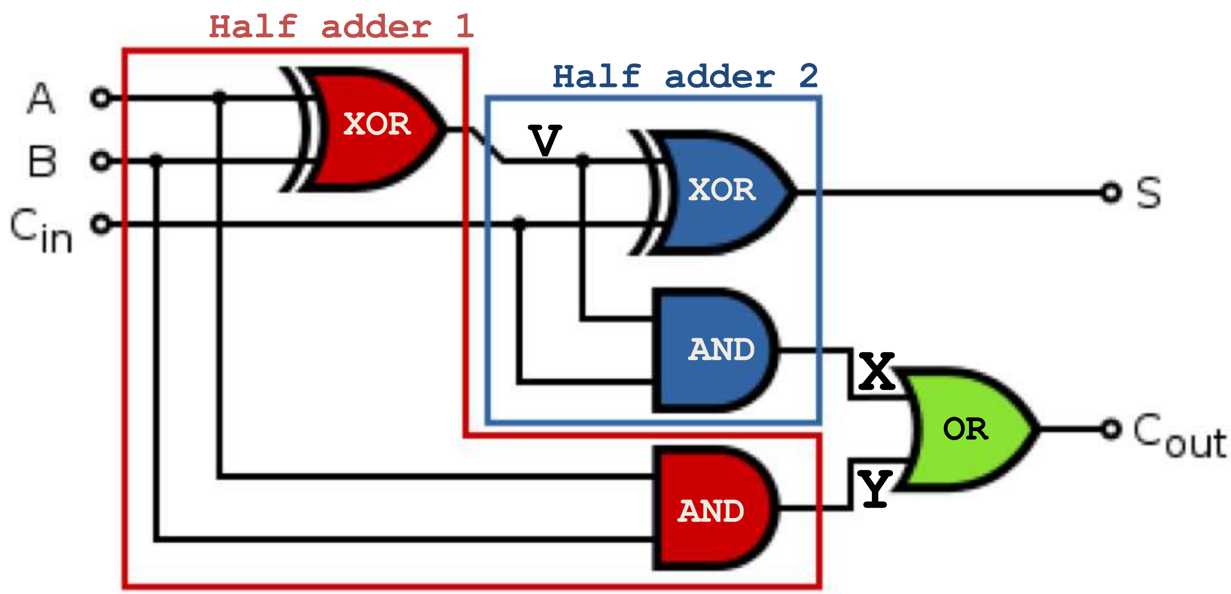
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Sum $= A \oplus B$
Carry $= A.B$



Two Half Adders = A Full Adder

- We could combine two half adders to make a full adder (which accepts 3 inputs: A, B and a carry-in)



- Full adders can be cascaded together to make a parallel adder that can add multi-bit binary numbers

Full Adder Truth Table

A	B	C_{in}	**V** **$A \oplus B$**	**sum** **S** **$V \oplus C_{in}$**	**X** **$V \bullet C_{in}$**	**.Y** **$A \bullet B$**	**carry** **C_{out}** **X + Y**
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

$V = A \oplus B$

$X = V \,.\, C_{in}$

$Y = A \,.\, B$

$S = V \oplus C_{in}$

$C_{out} = X + Y$

Full Adder Truth Table (Solution)

A	B	C_{in}	V $A \oplus B$	sum S $V \oplus C_{in}$	X $V \bullet C_{in}$	Y $A \bullet B$	carry C_{out} $X + Y$
0	0	0	*0*	*0*	*0*	*0*	*0*
0	0	1	*0*	*1*	*0*	*0*	*0*
0	1	0	*1*	*1*	*0*	*0*	*0*
0	1	1	*1*	*0*	*1*	*0*	*1*
1	0	0	*1*	*1*	*0*	*0*	*0*
1	0	1	*1*	*0*	*1*	*0*	*1*
1	1	0	*0*	*0*	*0*	*1*	*1*
1	1	1	*0*	*1*	*0*	*1*	*1*

Sanity check

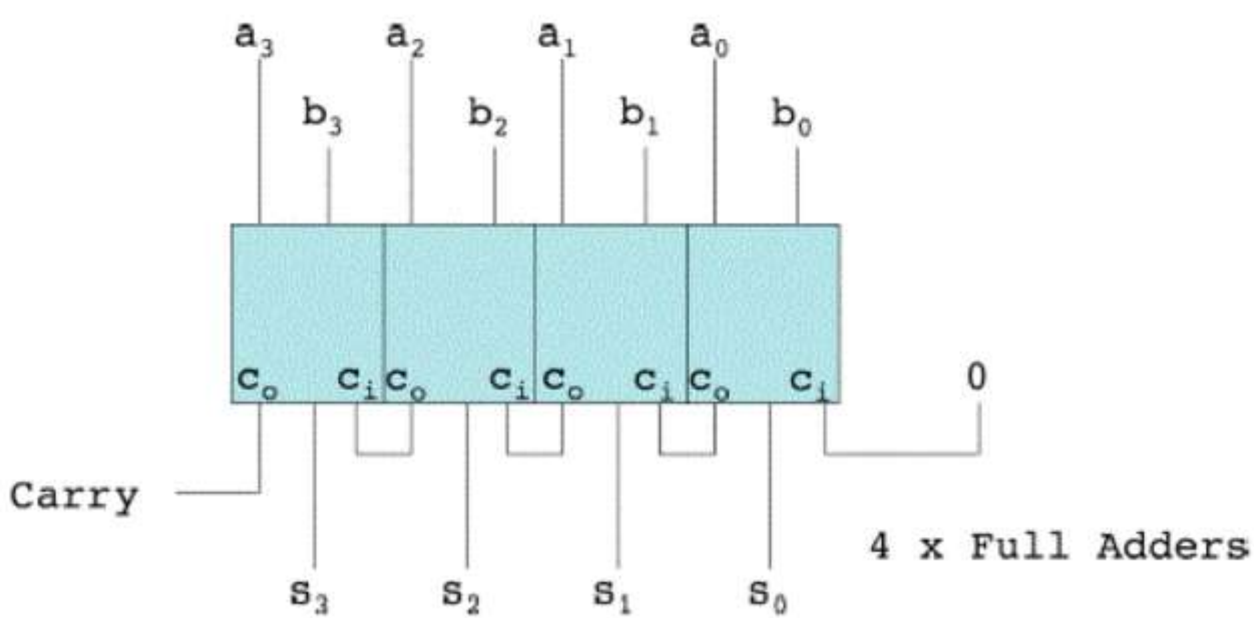
$S = 1$, whenever $A+B+C_{in} = 1$ or 3, else 0

$C_{out} = 1$, whenever $A+B+C_{in} = 2$ or 3, else 0

Parallel Adders

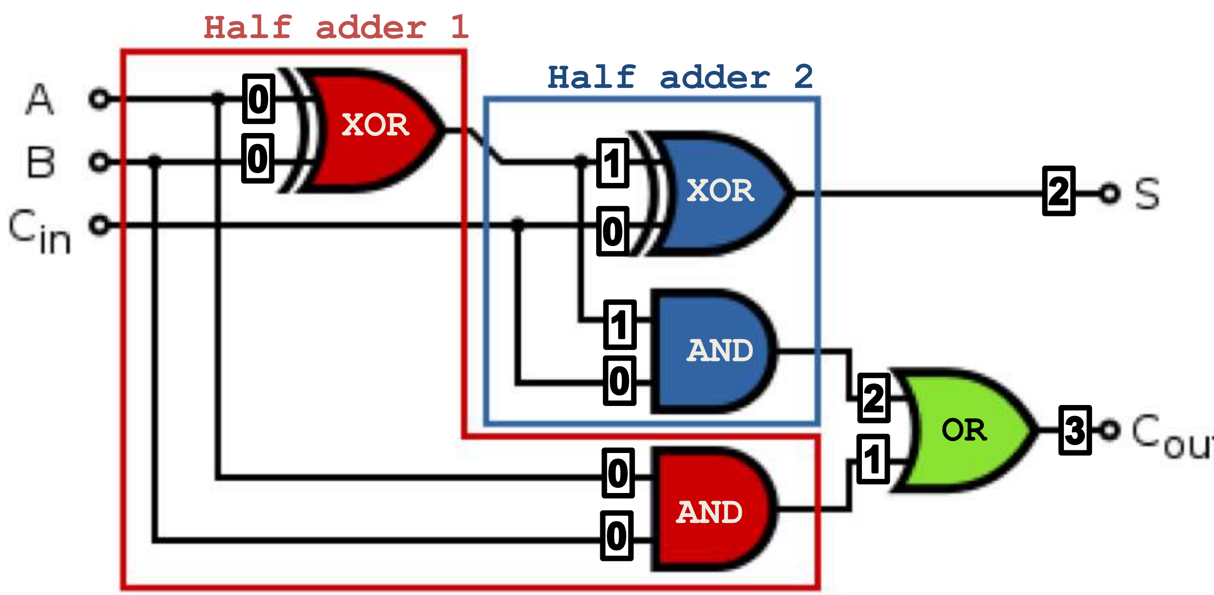
Parallel adder example

- 4 cascaded full-adders to add two 4-bit binary numbers
- the carry-out of each full-adder feeds into carry-in of the next full-adder
- the first carry-in is set to 0



Propagation Delay

- Notice how in the full adder below, a signal can pass through 1, 2 or 3 gates before reaching C_{out}.
  - Thus, if A,B,C_{in} change, it may take up to 3 cycles before S and C_{out} take their correct values.
  - This is in general called propagation delay and has to be taken into account when building circuits.



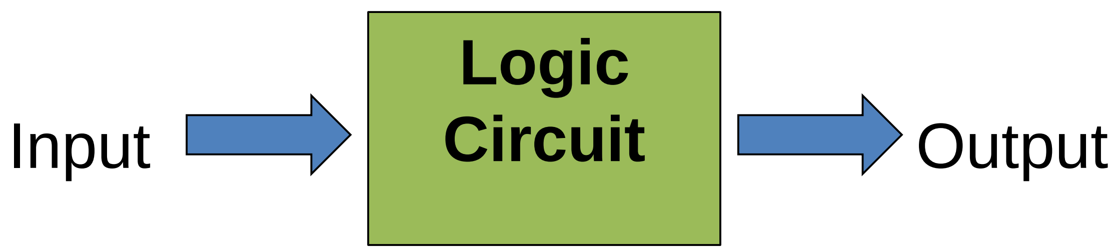
  - So it can take up to 3 clock ticks for the output to correctly reflect the input

Combinational Circuits

- A ***combinational circuit*** is one where the output state should instantly change whenever the input changes.
  - The full adder circuit is a combinational circuit.
  - Mathematically it is equivalent to the statement

    1 + 3 = 2 + 2, or
    A + 5 = B + 3, when A = 3 and B = 5

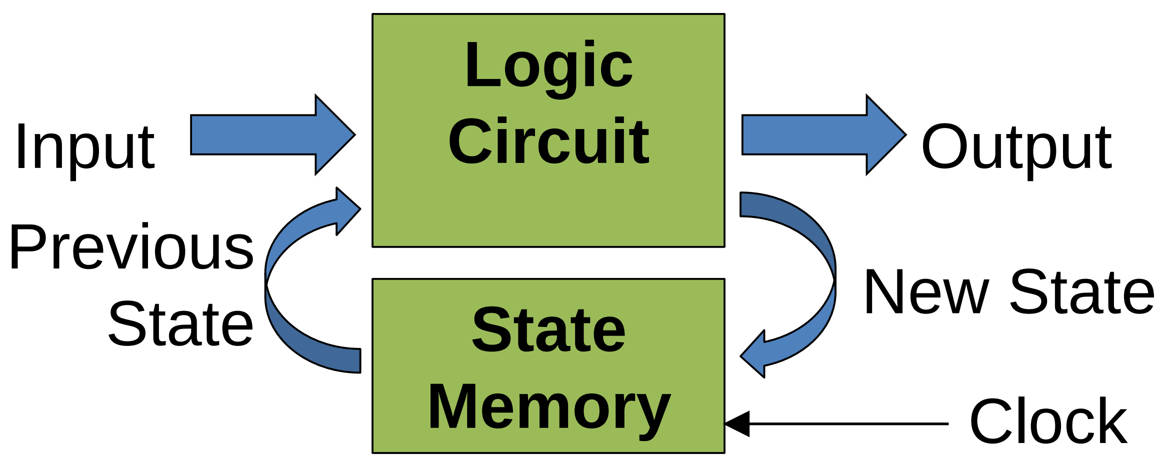
    Which is always true. The symbol A is ***substituted*** the number 3, and so for B.
- Clearly the output is a direct function of the input



Sequential Circuits

- A ***sequential circuit*** tis one that has some form of memory. It remembers its previous state, and this can affect its new state.
  - In practice, due to propagation delay, which can be seen as a form of memory, most circuits are treated as if they are sequential, even when they are not.

- **We will return to sequential logic in a future lecture.**



Python Code

- Consider the following Python code

```
def avg(a,b,c):
        return (a+b+c) / 3
```

- Clearly, the function add() need not remember what it did the last time. It just adds the three values. This is the ***combinational circuit.***

- Now consider the following

```
>>> avgbuf = [0, 0]
>>> def runavg(x):
      global avgbuf
      s = x + avgbuf[0] + avgbuf[1]
      avgbuf[0] = avgbuf[1]
      avgbuf[1] = x
      return s / 3

>>> runavg(3)
1
>>> runavg(3)
2
>>> runavg(3)
3
>>> avgbuf
[3, 3]
```

The program remembered the values it was given and so the answer depends on what happened before.

The is equivalent to a ***sequential circuit.***

Bit Masking

- Bit masks uses Boolean operations to access individual bits from binary data

- Used in ‘low level’ programming
  - device drivers and other hardware configuration/communication
  - data packet encoding/decoding
  - low level graphics

Bit Masking (Set)

- To set (i.e. make 1) a bit in a byte, OR it with a mask of all 0's except the bit to set
  - e.g. set b4 of 10001101

    10000101

    00001000 ← turn ON bits where there is a '1'

    ------------- Boolean *OR*

    10001101

- In Java:

```
short data = 0x85, maskON = 0x08;
short result = (short) (data | maskON);     // = 0x8D
```

- In Python:

```
data = 0b10000101
maskON = 0b00001000
result = data | maskON                      # = 0b10001101
```

Bit Masking (Reset)

- To reset (i.e. make 0) a bit in a byte, AND it with a mask of all 1's except the bit to reset
  - e.g. reset the LSB of 10001101

    10001101

    11111110 ← turn OFF bits where there is a '0'

    ------------- Boolean *AND*

    10001100

- In Java:

```
short data = 0x8D,    maskOFF = 0x01;
short result =  (short) (data & ! maskOFF);   // = 0x8C
```

- In Python:

```
data = 0b10000101
maskOFF = 0b00000001
result = data & ~maskOFF                         # = 0b10001100
```

Bit Masking (Flip)

- To flip (i.e. 1→0,0→1) a bit in a byte, XOR it with a mask of all 0’s except the bit(s) to flip
  - e.g. flip b4 and the LSB of 10001101

    10000101
    00001001 ← complement bits where there is a '1'
    ------------- Boolean *XOR*
    10001100

- In Java:

```
short data = 0x85, maskFLIP = 0x08 | 0x01;
short result =  (short) (data ^ maskFLIP);          // = 0x8d
```

- In Python:

```
data = 0b10000101
maskFLIP = 0x08 ^ 0x01
result = data ^ maskON                      # = 0b10001100
```

Quiz

- Given the equations for a full adder

$$S = (A \oplus B) \oplus C_{in}$$
$$C_{out} = (A \oplus B)\, C_{in} + AB$$

Show that when A = /B, then

$S = /C_{in}$ always, and

$C_{out} = C_{in}$ always

using truth tables.

- **If you were to use such a circuit fin a parallel adder or 8-bit integers:**
    - **What is it useful for?**
    - **What does it do?**
  - **When Cin = 0,**