

Advanced Lab 1

Transistors and Bitwise Operators

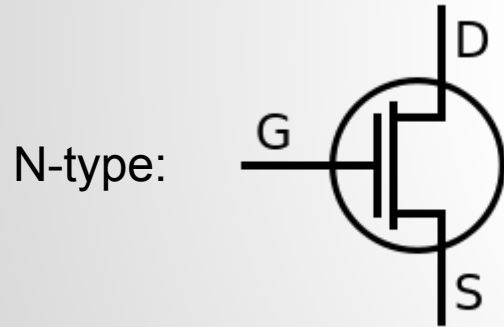
Andrew Wilder

Topics

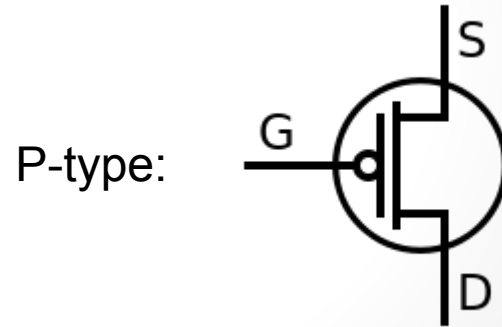
- Transistors
 - Fundamentals
 - From Truth Tables to Circuits
- Bitwise Operators
 - How They Work
 - Mask Generation
 - Optimizing with Bitwise Operators

Transistors: Fundamentals

There are two types of transistors:



Attached to ground

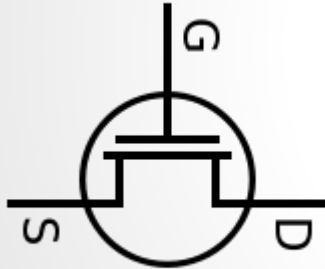


Attached to power

Transistors: Fundamentals

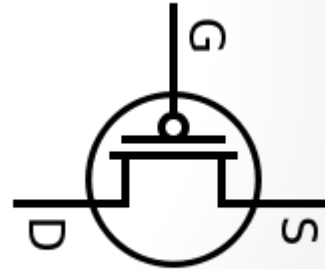
The base allows a signal through based on its value:

N-type:



Allow a signal through when
base == 1

P-type:

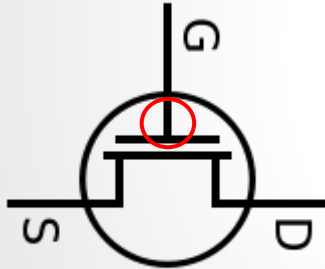


Allow a signal through when
base == 0

Transistors: Fundamentals

My memorization mnemonic:

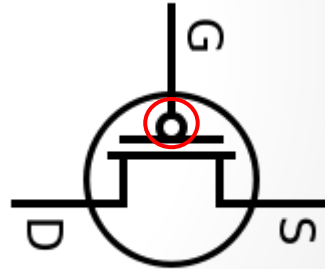
N-type:



Looks like a 1

Allow a signal through when
base == 1

P-type:

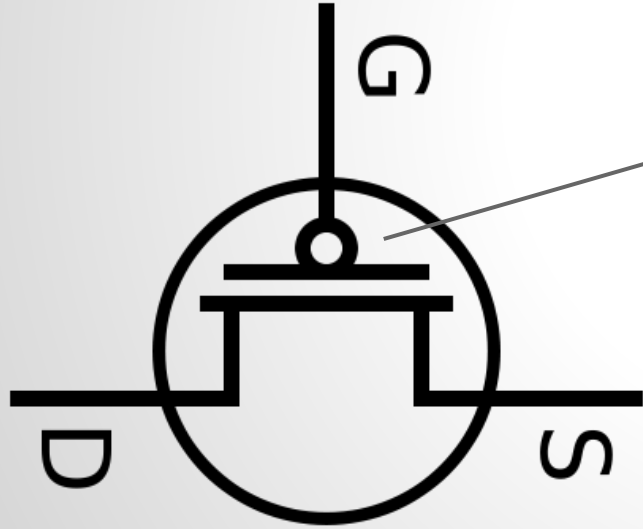


Looks like a 0

Allow a signal through when
base == 0

Transistors: Fundamentals

My memorization mnemonic:



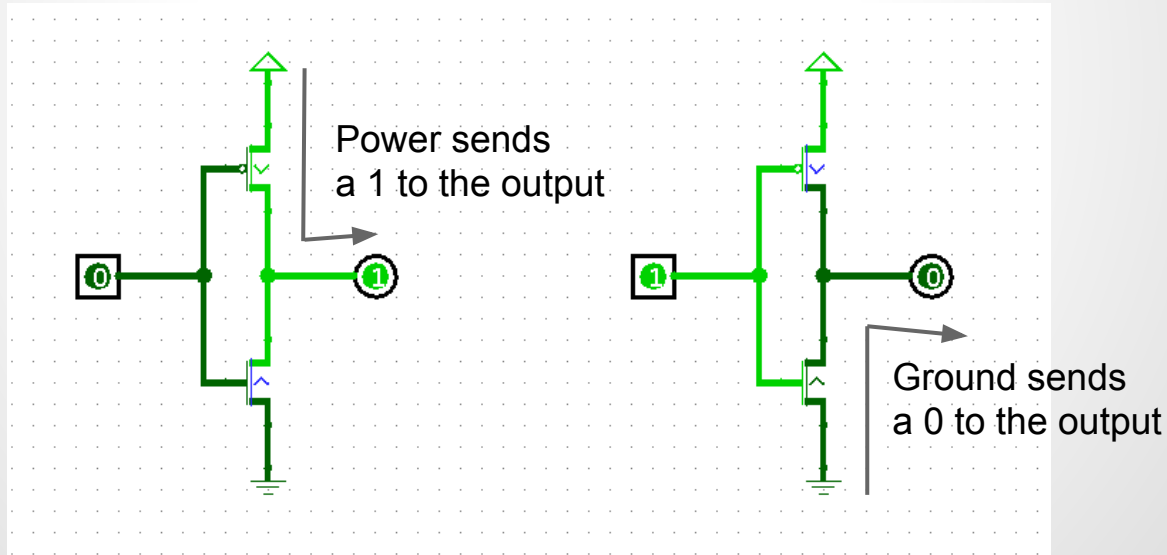
Has a loop like a **P**

P-type = **Power**

(hook these up to power)

From Truth Tables to Circuits

Forget any science you've learned about electrons traveling from power to ground; for the purpose of practicing with transistor logic, think of each as simply “sending” a logical signal to the output:



From Truth Tables to Circuits

The easiest way to figure out how to wire a NAND or NOR gate is to construct a truth table. For example, here is the truth table for NAND:

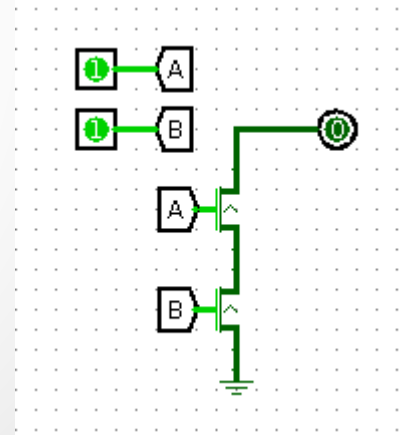
| A | B | Out |
|---|---|-----|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

From Truth Tables to Circuits

Find the unique output for the truth table, and wire that first in series. The reason for this is that a series circuit requires that **all** the conditions be true:

| A | B | Out |
|---|---|-----|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Only when all the conditions, that is, $A == 1$ and $B == 1$, will the output be zero, so use a series of transistors

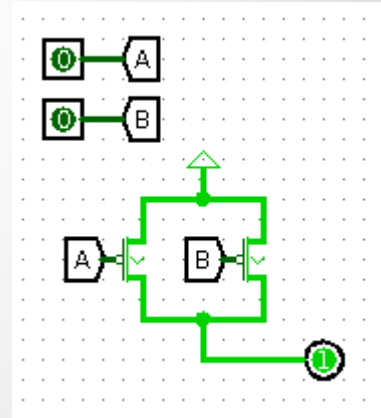


From Truth Tables to Circuits

Next, wire all the other outputs in parallel. We use parallel because parallel transistors simply allow **any** of the conditions to be true:

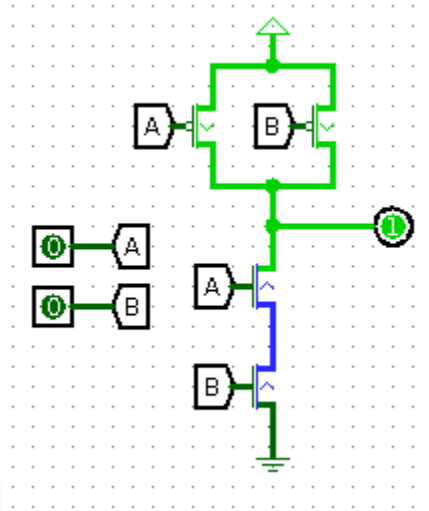
| A | B | Out |
|---|---|-----|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Output a 1 if either $A == 0$, $B == 0$, or both are 0, so use parallel transistors



From Truth Tables to Circuits

Combine the two parts together for the completed NAND:



From Truth Tables to Circuits

Let's design a circuit for A NOR B NOR C!

| A | B | C | Out |
|---|---|---|-----|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

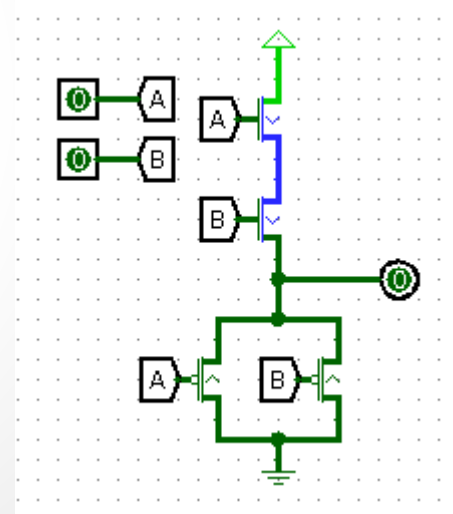
From Truth Tables to Circuits

Caveat: AND, OR - is this correct?

| (AND) | | |
|-------|---|-----|
| A | B | Out |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Unique case: Series

Non-unique case: Parallel



From Truth Tables to Circuits

Caveat: AND, OR - is this correct?

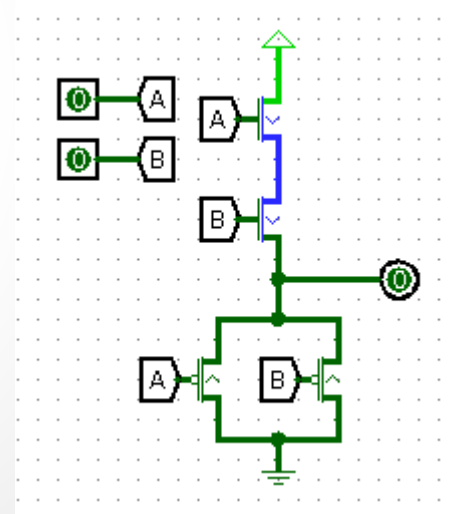
NO!

| (AND) | | |
|-------|---|-----|
| A | B | Out |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**P-type should connect
to power, N-type to ground**

Unique case: Series

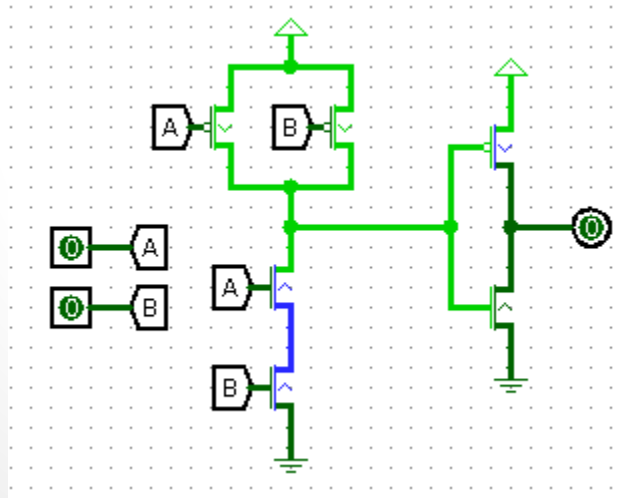
Non-unique case: Parallel



From Truth Tables to Circuits

Caveat: AND, OR

[AND, OR] should be implemented with [NAND, NOR] + NOT



Bitwise Operators: How They Work

Bitwise operators are fast, simple operations that can be performed on sets of bits.

& bitwise AND

| bitwise OR

~ bitwise NOT

<< >> shift operator

^ bitwise XOR (can be made with AND, OR, NOT)

Bitwise Operators: How They Work

~ (Flip bits)

The ~ operator can be used to flip the bits in a number.

```
~  0100
-----
   1011
```

Bitwise Operators: How They Work

| (Set bits)

The | operator can be used with a mask to set bits.

```
      0100
    | 0010
    -----
      0110
```

Bitwise Operators: How They Work

& (Reset bits)

The & operator can be used with a mask to reset bits.

$$\begin{array}{r} 1011 \\ \& 1101 \\ \hline 1001 \end{array}$$

Bitwise Operators: How They Work

^ (Toggle bits)

The ^ operator can be used with a mask to toggle bits.

$$\begin{array}{r} 1010 \\ ^ 0110 \\ \hline 1100 \end{array}$$

Bitwise Operators: How They Work

<< or >> (Shift bits)

The shift operators << or >> can move bits left or right.

00011010 << 2

01101000

The diagram illustrates a left shift by 2 bits. The original 8-bit value 00011010 is shown in red. Three arrows point from the third, fourth, and fifth bits (0, 1, 1) to the third, fourth, and fifth positions of the result 01101000. The first two positions of the result are filled with zeros. The last two bits of the result (0, 0) are shown in blue.

Zeros are shifted in from the right

10110100 >> 2

11101101

The diagram illustrates a right shift by 2 bits. The original 8-bit value 10110100 is shown in red. A large arrow points from the first bit (1) to the first position of the result. Three arrows point from the second, third, fourth, and fifth bits (0, 1, 1, 0) to the second, third, fourth, and fifth positions of the result. The first two positions of the result (1, 1) are enclosed in a blue box. The last three bits of the result (1, 0, 1) are shown in red.

MSB is shifted in from the left

Bitwise Operators: Mask Generation

Need a single 1 at location n to set or toggle a bit?
Shift a 1 over to the location:

00000001 << 3

00001000

Bitwise Operators: Mask Generation

Need a single 0 at location n to reset a bit?
Shift a 1 over to the location, and flip the bits:

00000001 << 3

00001000 ~

11110111

Bitwise Operators: Mask Generation

Case study: get_a_byte

Problem: We want to get a single byte from a 32-bit integer value, given an index 0 - 3 for which byte.

| | | | |
|----------|----------|----------|----------|
| 01001101 | 00111010 | 01010000 | 10110101 |
| ----- | ----- | ----- | ----- |
| B3 | B2 | B1 | B0 |

Bitwise Operators: Mask Generation

Case study: get_a_byte

Answer: Shift the byte over to the least significant 8 bits (which = 2):

```
01001101 00111010 01010000 10110101
00000000 00000000 01001101 00111010
```

But there's still some garbage above the desired byte...

Bitwise Operators: Mask Generation

Case study: get_a_byte

Mask using & to remove undesired bits:

```
00000000 00000000 01001101 00111010
& 00000000 00000000 00000000 11111111
-----
00000000 00000000 00000000 00111010
```

Bitwise Operators: Mask Generation

Case study: get_a_byte

What this looks like in code:

```
return (n >> (which * 8)) & 0xFF;
```

Shift the number...



...in increments of 8 bits...



...and mask the result.



Optimizing with Bitwise Operators

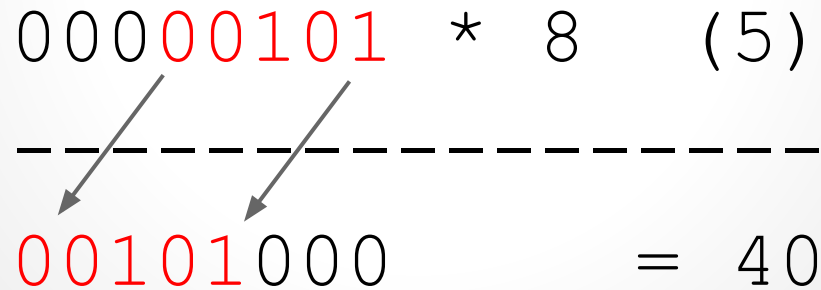
Powers of 2 are very useful!

- Operating system uses them for page size and buddy allocators (more on that later).
- Cryptography uses them to bound key and cipher block sizes.
- You can use them to efficiently perform multiplication, division and modulus!

Optimizing with Bitwise Operators

Multiplication:

When you multiply by a power of 2, you essentially shift the bits left:

$$\begin{array}{r} 00000101 * 8 \quad (5) \\ \hline 00101000 = 40 \end{array}$$


Optimizing with Bitwise Operators

Multiplication:

When you multiply by a power of 2, you essentially shift the bits left:

00000101 $\ll 3$ (5)

00101000 = 40

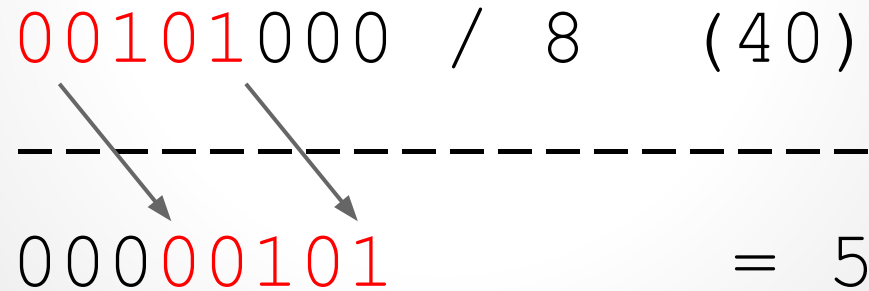
The diagram illustrates the bit shift operation. The top line shows the binary number 00000101, where the last four bits (0101) are highlighted in red. To the right of this number is a green box containing the operation $\ll 3$, followed by (5). Below the top line is a dashed horizontal line. Two arrows point from the red-highlighted bits of the top line to the corresponding bits in the bottom line. The bottom line shows the result 00101000, where the first four bits (0010) are highlighted in red, and followed by = 40.

It's the same as left shifting by $\log_2(n)$!

Optimizing with Bitwise Operators

Division:

When you divide by a power of 2, you essentially shift the bits right:

$$\begin{array}{r} 00101000 \text{ / } 8 \quad (40) \\ \hline 00000101 \quad = 5 \end{array}$$


Optimizing with Bitwise Operators

Division:

When you divide by a power of 2, you essentially shift the bits right:

00101000 >> 3 (40)

00000101 = 5

It's the same as right shifting by $\log_2(n)$!

Optimizing with Bitwise Operators

Modulus:

When you mod by a power of 2, you zero out all the bits above the first $\log_2(n)$ bits:

00110110 % 16 (54)
 ↓ ↓
 0000110 = 6

Optimizing with Bitwise Operators

Modulus:

When you mod by a power of 2, you zero out all the bits above the first $\log_2(n)$ bits:

The diagram illustrates the modulus operation $10 \& 15$. The first row shows the binary representation of 10 as 00110110, followed by the operation $\& 15$ (where 15 is in a green box) and the decimal value (54). The second row shows the result of the operation: 00000110, followed by an equals sign and the decimal value 6. Arrows point from the first four bits of the first row to the first four bits of the second row, indicating that the higher-order bits are zeroed out.

$$\begin{array}{r} 00110110 \quad \boxed{\& 15} \quad (54) \\ \hline 00000110 \quad = 6 \end{array}$$

It's the same as bitwise AND with $(n-1)$!

Optimizing with Bitwise Operators

Case study: get_a_byte

Let's return to our old code:

```
return (n >> (which * 8)) & 0xFF;
```

How can we improve this line?

Optimizing with Bitwise Operators

Case study: get_a_byte

Let's return to our old code:

```
return (n >> (which * 8)) & 0xFF;
```

How can we improve this line? **Use the shift operator!**

```
return (n >> (which << 3)) & 0xFF;
```

Optimizing with Bitwise Operators

Case study: power_of_2

Look at these powers of 2:

00100000 = 32

00001000 = 8

01000000 = 64

Notice a pattern?

Optimizing with Bitwise Operators

Case study: power_of_2

Look at these powers of 2:

00100000 = 32

00001000 = 8

01000000 = 64

Notice a pattern?

Only one bit set.

Optimizing with Bitwise Operators

Case study: power_of_2

How can we determine if only one bit is set?

```
int set = 0;
for(int i = 0; i < 31; ++i)
    if(n & (1 << i) != 0)
        ++set;
return n > 0 && set == 1;
```

Can we do better than a loop?

Optimizing with Bitwise Operators

Case study: power_of_2

Something else special about powers of 2:

For any power of 2 “n”, the only bit activated in n is farther left than all the activated bits of values less than n. This property only applies to powers of 2!

| | | | | | |
|-----|---|---------|-----|---|----------|
| n | : | 0010000 | m | : | 00011010 |
| n-1 | : | 0001111 | m-1 | : | 00011001 |

These bits didn't change!

Optimizing with Bitwise Operators

Case study: power_of_2

n : 0010000

n-1 : 0001111

Since these values do not share any activated bits, a bitwise AND operation will produce 0 for powers of 2!

$$16 \ \& \ 15 = 0$$

16 is a power of 2

$$26 \ \& \ 25 = 24$$

26 is not a power of 2

Optimizing with Bitwise Operators

Case study: power_of_2


What this looks like in code:

```
return n > 0 && (n & (n-1)) == 0;
```

Number must be positive



Power of 2 property



This is sometimes an interview question! Know how to do it!