Forming the truth table without learning it by heart and then using it to manipulate bits

Truth tables can become lengthy and complex. They are also rather far from the way that we usually express ourselves. The implication of this is that remembering by heart all the alternatives that build up a truth table is difficult. Perhaps, it is easier to come up with a method to construct it by considering the first principles behind it and reflecting on the name of each operator. In this short discussion, we'll consider variables p and q.

Conjunction, or logical AND, denoted with ^

The point to remember about AND is that the result is set (logical 1) only when p **AND** q are **both** set (logical 1). In all other cases, it is cleared (logical 0). So, we fill the truth table with all options but only assign logical 1 to one of them:

p	q	p ^ q
1	1	1
1	0	0
0	1	0
0	0	0

Logical AND can be used in bitwise operations to **clear** bits. Assuming that we have a byte A some of whose bits need to be cleared, the process is as follows:

1. Use a second byte as a mask.
2. Based on the above we see that all bits that need to be cleared need to be ANDed with binary 0. Clear all the relevant bits in the mask to force the bits of interest to 0.
3. Perform a logical AND between byte A and the mask byte, placing the result in A.

Example: Assuming that we have a byte A = 0b01110011 and we need to clear bits 5 & 1. The process is as follows:

1. Form a mask byte with all bits **set**.
2. **Clear** bits 5 and 1 in the mask byte.
3. **AND** byte A with the mask:
4. Store the result in A

A = 0b01110011
mask = 0b11011101

A ^ mask = 0b01110011 ^ 0b11011101 = 0b01010001

Bits 5 and 1 have been cleared.

Disjunction, or logical OR, denoted with v

The point to remember about OR is that the result is set (logical 1) when **EITHER** p **OR** q are set (logical 1). So, we fill the truth table with all options and assign T to any row in which p or q is set:

p	q	p v q
1	1	1
1	0	1
0	1	1
0	0	0

Logical OR can be used in bitwise operations to **set** bits. Assuming that we have a byte A some of whose bits need to be set, the process is as follows:

1. Use a second byte as a mask.
2. Based on the above we see that all bits that need to be set need to be ORed with binary 1. Set all the relevant bits in the mask to force the bits of interest to 1.
3. Perform a logical OR between byte A and the mask byte, placing the result in A.

Example: Assuming that we have a byte A = 0b01110011 and we need to set bits 7 & 2. The process is as follows:

1. Form a mask byte with all bits **cleared**.
2. **Set** bits 7 and 2 in the mask byte.
3. **OR** byte A with the mask:
4. Store the result in A

A = 0b01110011
mask = 0b10000100

A v mask = 0b01110011 ^ 0b10000100 = 0b11110111

Bits 7 and 2 have been set.

Exclusive XOR, denoted with ⊕

The point to remember about XOR is that it is set (logical 1) when **ONLY ONE** of p or q is set (logical 1). So, we fill the truth table with all options and assign T to any row in which only p or only q are set:

p	q	$p \oplus q$
1	1	0
1	0	1
0	1	1
0	0	0

Logical XOR can be used in bitwise operations to **toggle** bits. Assuming that we have a byte A some of whose bits need to be toggled, the process is as follows:

1. Use a second byte as a mask.
2. Based on the above we see that all bits that need to be toggled need to be XORed with binary 1. Set all the relevant bits in the mask to toggle the bits of interest.
3. Perform a logical XOR between byte A and the mask byte, placing the result in A.

Example: Assuming that we have a byte A = 0b01110011 and we need to toggle bits 7 & 1. The process is as follows:

1. Form a mask byte with all bits **cleared**.
2. **Set** bits 7 and 1 in the mask byte.
3. **XOR** byte A with the mask:
4. Store the result in A

A = 0b01110011
mask = 0b10000010

A ⊕ mask = 0b01110011 ^ 0b10000010 = 0b11110001

Bits 7 and 1 have been toggled.