Простейшие логические функции

Итак, мы выяснили, что логические сигналы объединяются друг с другом при помощи логических функций. На самом деле, я уже упоминал это в статье про ПЛИС о том, что всё делается совершенно аналогично программированию. Давайте глянем ещё раз:

Провод = Переменная типа BOOL
Логическая функция = Функция языка, применимая к переменным типа BOOL

Поэтому ничего не боимся и читаем совершенно спокойно. Ничего нового не будет, будет просто чуть другое представление уже известного. Но всё-таки я буду давать чуть более расширенные пояснения, чем просто аналогичные функции языка Си.

Элемент ИЛИ

Итак, начнём с элемента ИЛИ. Он эквивалентен оператору «||» языка Си. На выходе элемента ИЛИ будет логическая единица в случае, если хотя бы на одном входе имеется логическая единица. Если на всех входах логические нули – на выходе будет также логический ноль. При работе с логическими элементами, принято пользоваться таблицами истинности. Не будем нарушать это правило, и нарисуем несколько таких таблиц. Функция ИЛИ с двумя входами называется 2ИЛИ и описывается таблицей истинности:

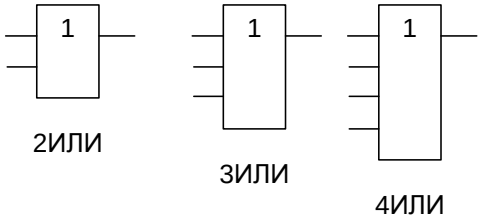
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Страшно? По-моему, нет Давайте для пущей храбрости нарисуем таблицу истинности для функции 3ИЛИ

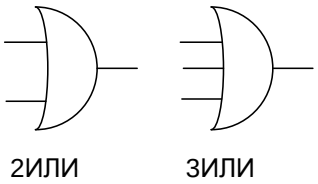
X1	X2	X3	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Чтобы суть элемента ИЛИ лучше запомнилась, давайте рассмотрим аналогию: Охранные системы. Какой бы из датчиков (датчиков движения, пожарных датчиков, прочих датчиков) ни сработал бы, всё равно надо поднимать шум и будить охранника. Вот это как раз и выполняется по правилам функции ИЛИ. Количество входов – столько, сколько датчиков. Выход – один. Собственно, в функциях языка С выход тоже один, и он называется результатом функции.

А как такие функции обозначаются на схемах? По ГОСТ – так:



По стандарту ANSI – примерно так (извините за корявость, но я не рисую по ANSI, но на схемах могут встречаться, не надо пугаться):



Входы ВСЕГДА рисуют слева, выходы – справа. Некоторые считают, что иногда будет красивее нарисовать наоборот, входы справа, выход – слева (это если разъёмы на картинке так расположены). Так вот. Это неправильно. Без этого ВСЕГДА можно красиво обойтись, если чуть-чуть подумать (равно как в языке Си можно обойтись без меток).

Элемент И

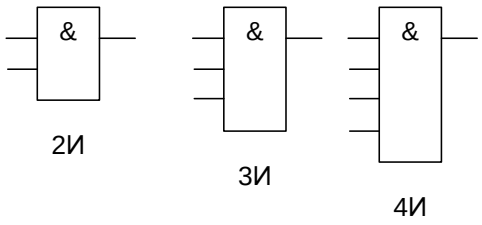
Элемент И эквивалентен оператору «&&» языка Си. То есть, на выходе будет логическая единица тогда и только тогда, когда на ВСЕХ входах имеется логическая единица. Нарисуем таблицы истинности для элементов 2И и 3И

X1	**X2**	**Y**
0	0	0
0	1	0
1	0	0
1	1	1

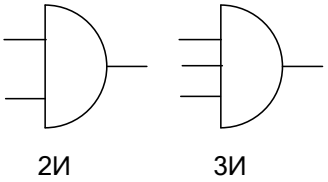
X1	**X2**	**X3**	**Y**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Жизненная аналогия – поезд. Пока проводники всех вагонов не дадут добро, машинист не начнёт движение состава.

Теперь обозначение по ГОСТ:



Ну, и в меру корявое обозначение по ANSI:

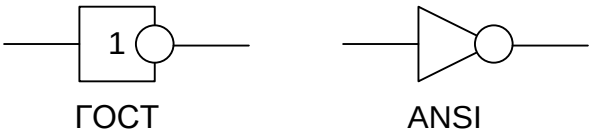


Элемент НЕ

А вот с элементом НЕ мы уже познакомились ранее. Его аналог на Си: «!», таблица истинности у него проста до ужаса:

X1	**Y**
0	1
1	0

Обозначение же у него такое (признак инверсии – кружочек, он нам ещё много где встретится):



Комбинированные функции

Вообще, перечень функций при классическом подходе жёстко лимитирован тем, что продаётся в магазине. Какие бывают типы микросхем – смотрим в справочниках (в доинтернетовское время – страшный дефицит, обладание справочником Шило было чем-то вроде… Вроде… Вроде обладания футболкой «АстроСофт», не всем это было под силу). Если мы откроем справочник, то увидим элементы 2И, 3И, 4И, 8И, 2ИЛИ, 4ИЛИ, 8ИЛИ… А также всякие там 2И-НЕ, 4ИЛИ-НЕ и т.п. Что это такое? Это чтобы нам сэкономить число микросхем на плате, сделали микросхемы, где объединены две функции. Какая-либо с функцией НЕ.

Все уже догадались, как работают подобные функции, но традиции требуют привести таблицы истинности. Без них при описании логических функций никуда. Поэтому приведём парочку:

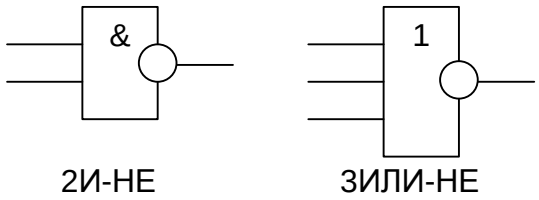
Для функции 2И-НЕ таблица истинности будет такая:

X1	X2	Y
0	0	1
0	1	1
1	0	1
1	1	0

А, скажем, для 3ИЛИ-НЕ – такая:

X1	X2	X3	Y
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

В обозначении же, просто добавляется уже знакомый нам кружок:



Теорема Де Моргана

После того, как лекторы рассказывают о функциях 2что_нибудь-НЕ, они всегда рассказывают и теорему Де Моргана. Давайте не будем нарушать традиций, тем более, что она поможет нам закрепить навыки работы с таблицами истинности, а эти навыки нам ещё пригодятся.

Классическая запись теоремы выглядит так:

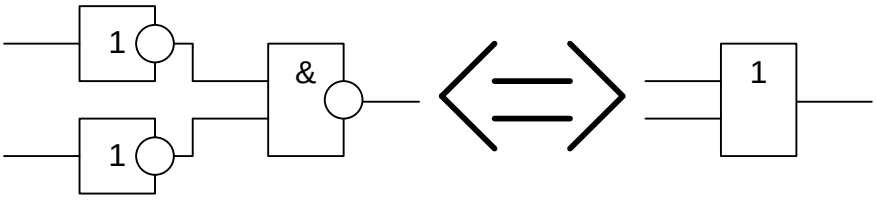
$$\overline{X1} \ \& \ \overline{X2} = \overline{X1 \mid Y2}$$

Где надчёркивание – это обозначение инверсии

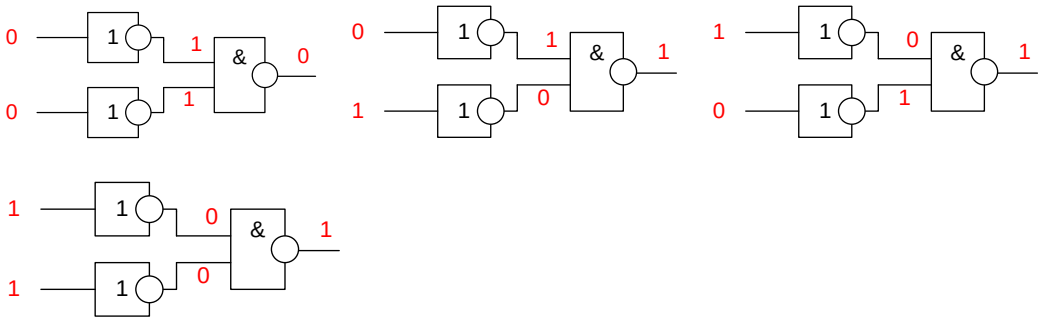
НА языке Си это будет выглядеть так:

```
!((!x1) && (!x2)) == (x1 || x2);
```

А в виде схемы – так:



Давайте докажем эту теорему. Построим таблицу истинности для левой части:



Итого получаем:

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Ну, а таблицу для правой части мы уже строили, она точно такая же, что и требовалось доказать.

Это правило было чрезвычайно важно при работе с классическими микросхемами, когда нарисуешь простыню и думаешь: «Мам моя, где ж я столько накуплю». И вдруг видишь, что если здесь добавить инвертор (который случайно остался в составе неполностью используемой микросхемы), то можно поменять тип элемента и выкинуть целую микросхему (у меня реально были такие ситуации). Во времена ПЛИС, за нас всё это сделает оптимизатор, но доказывая теорему, мы попрактиковались с построением таблиц истинности, уже хлеб. А на закуску, предлагаю вам самостоятельно доказать вторую теорему Де Моргана, которая на языке Си записывается так: !((!x1) || (!x2)) == (x1 && x2);

Практический пример – кодовый замок

Классическая реализация

Ну что. И напоследок разбавим наше сухое повествование каким-нибудь примером, который использовал бы описанные логические элементы. В ЦО раньше на дверях был механический кодовый замок, в Новгородском филиале такие до сих пор используются. Там надо нажать сразу несколько кнопок, тогда он откроется. Если хоть одна кнопка неверна, он не откроется. Давайте сделаем такой же, но электронный.

Сначала я спроектирую его в таком базисе, какой можно сделать на реально имеющихся микросхемах.

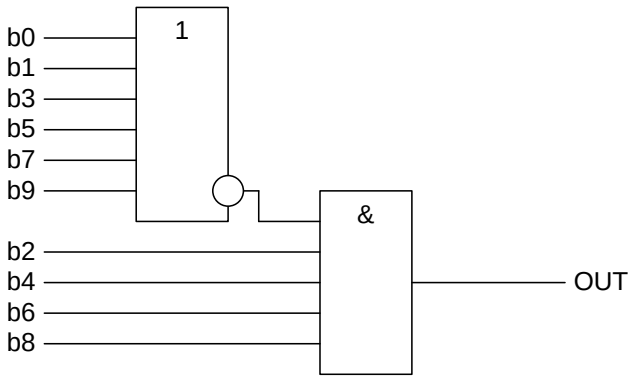
Пусть наш замок открывается, если одновременно нажаты кнопки 2, 4, 6 и 8. Я не буду рассуждать о том, как правильно выбирать логические элементы по справочнику (почему – объясню в следующем разделе), я просто выберу их на основании своего опыта.

Что такое замок? Это функция, на входе которой имеется десять переменных типа BOOL – b0, b1, b2, …, b9. Каждая переменная равна 0 (FALSE), если кнопка не нажата и 1 (TRUE), если нажата. И из функции есть выход, который содержит 0, если комбинация не верна и 1 – если верна. С точки зрения электроники, если на выходе появилась единица, то сработал электромагнит и всё нам открыл.

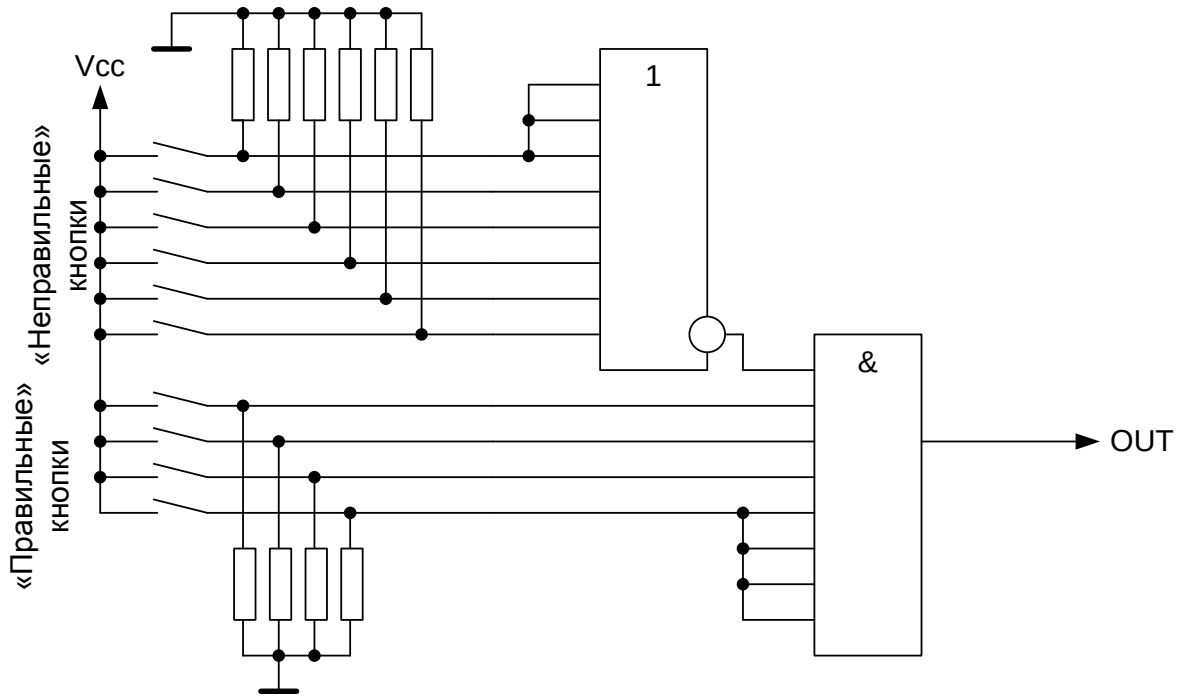
Сначала запишем это на привычном нам языке Си:

```
BOOL lock (BOOL b0,BOOL b1, BOOL b2, BOOL b3,…, BOOL b9)
{
	// Если нажаты правильные кнопки
	BOOL right = b2 && b4 && b6 && b8;

	// Если нажата хоть одна неправильная кнопка
	BOOL wrong = b0 || b1 || b3 || B5 || b7 || b9;

	return (right && (!wrong));
}
```

Вроде так? А теперь барабанная дробь, рисуем то же самое в виде схемы и удивляемся, насколько всё аналогично…



И, собственно, всё. Кнопки я не нарисовал для простоты… А так – схема готова. Если уж кто хочет совсем реальную схему, то она будет выглядеть так:



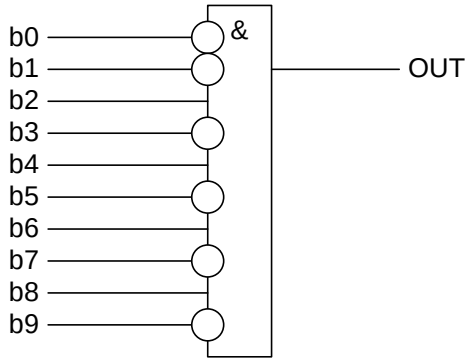
На реальной схеме мы видим два отличия от прошлой. Во-первых, почему-то функции стали восьмивходовыми. Всё просто – в реальной жизни не выпускается микросхем 6ИЛИ-НЕ и 5И. Поэтому пришлось взять такие, какие выпускаются, а все лишние выводы соединить так, чтобы не мешались. Ну, и резисторы. Помните смывной бачок? Я его во всех лекциях буду вспоминать, очень полезная штука. Эти резисторы обеспечивают подтяжку входов к общему проводу. В общем, если заряд случайно откуда-то взялся – он стечёт в канализацию. Но если кнопка нажата – логическая единица победит, так как кнопка – это широкая труба, а резистор – всё-таки тонкая трубочка. Но когда кнопку отпустят – остатки заряда вновь стекут через резистор, и на линии снова воцарится логический ноль…

Реализация на базе ПЛИС

Ура! Ура! Теперь я с чистой совестью могу повторить один из аргументов, которые приводил в статье про ПЛИС, но на этот раз те, кто читал статьи внимательно, уже всё поймут. Итак. Давайте запишем функцию замка на языке Си чуть иначе:

```
BOOL lock (BOOL b0,BOOL b1, BOOL b2, BOOL b3,…, BOOL b9)
{
	return (!b0) && (!b1) && (b2) && (!b3) && (b4) && (!b5) && (b6) && (!b7) && (b8) && (!b9);
}
```

Имеем право? Не вижу причин, почему нет. То есть, мы в самой функции указали, какие кнопки правильные, а какие – нет. Зарисуем это в виде схемы



Будет работать такая схема? Если мы построим таблицы истинности для прошлой и для этой, то они совпадут. Значит будет. А почему мы сразу не нарисовали её? А потому что максимальная микросхема, какая бывает – это 8И. 10 входов пришлось бы каскадировать, а ещё и входные инверторы пришлось бы брать внешние – уже три микросхемы (против двух в прошлой схеме) и три шага обработки (а значит – низкое быстродействие). В общем, при классическом подходе – неприемлемо… Но то при классическом.

Откройте статью про ПЛИС и посмотрите структуру простейшей ПЛМ. Что мы видим? Правильно, опциональные инверторы на входе, потом – элемент И (скажу, что даже в самой вшивой ПЛМ он 16-входовой). То есть, мы реализуем наш замок воспользовавшись одним элементом ПЛМ. А этих элементов там много. Так что ещё и останется. Я молчу про CPLD, где их ещё больше!

Что же до FPGA, то я тоже уже говорил, там компилятор рассчитывает таблицу истинности функций и кладёт её в ОЗУшки. Так что там всё тоже получится просто и симпатично. И займёт это – хорошо если 1% от ёмкости, а то и меньше.

Вот и получается, что в классическом случае, нам пришлось подгонять логику работы замка под имеющуюся комплектацию (хорошо, если она есть в магазине, а то потом ещё переделывать придётся), затем – купить эти две микросхемы (а у микросхем более 70% стоимости составляет корпус, так что чем их больше, тем дороже, ПЛИС может оказаться дешевле этой парочки), затем – как-то разместить их на плате и развести связи между ними… А в случае с ПЛИС, мы взяли самую понятную для нас функцию, а всё остальное за нас сделал компилятор. И не просто поместил всё в одну микросхему, а ещё и оставил в ней место для других функций. Как говорится, почувствуйте разницу. И забудьте о классическом подходе, как о страшном сне.

Функции для обработки данных (из которых потом соберём процессор ☺) рассмотрим чуть позже, а пока давайте изучим такую полезную вещь, как триггер. Но сделаем мы это уже в следующей лекции.