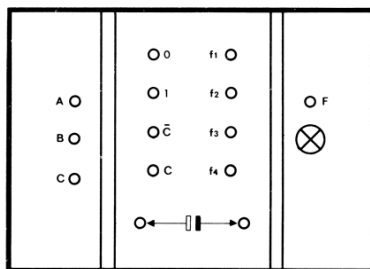


# Dissecting the CL1601 Logic Circuit

This well-conceived little circuit was the core building block of the Philips CL1601 computer kit. It is depicted (in a symbolic fashion) in Figure 1. The terminals **A**, **B**, and **C** near the left edge of the block accept the data input. The output is provided by terminal **F** near the right edge. The “inner” terminals are used to “program” the circuit as a specific logic function. The “instruction word” is actually a “nibble” – a 4-bit pattern applied to terminals f1-f4.

By hooking each of the terminals f1-f4 up to either terminal 1 (+) or terminal 0 (GND), we can instruct the circuit to emulate each of the 16 different operators possible for two binary variables.



A	B	F	
0	0		f1 =
0	1		f2 =
1	0		f3 =
1	1		f4 =

Table 1

Figure 1

How does this work? Well, let’s picture a generic truth table for two variables, A and B, with output F as displayed in Table 1. The desired binary function will correspond to a “pattern” in the F column. For example, A **AND** B corresponds to the pattern (0, 0, 0, 1), with row f1 represented by the first element of in the brackets, row f2 to the second one etc. A **XOR** B would correspond to (0, 1, 1, 0), A **EQUALS** B to (1, 0, 0, 1), and so on.

In Figure 1, you can make out the two terminals **0** and **1**. These are used to “program” the circuit. For logic functions with just two variables, the procedure is straightforward: just connect the respective f-terminal with the **F**-value defined for this row. For the **AND** function this would look as shown in Figure 2 and 3, which show two functionally equivalent alternatives for wiring up the circuit:

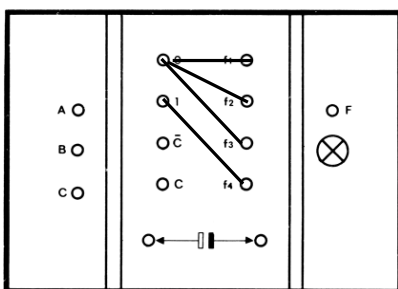


Figure 2

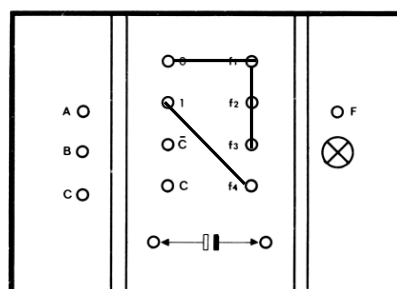


Figure 3

In slightly more abstract terms, we have 2 constant values (0 and 1) to initialize the variables f1, f2, f3, and f4. This yields  $2^4 = 16$  different combinations, the necessary and sufficient number of alternatives for encoding the full range of (2-valued) Boolean operators, whereof the unary functions (negation, identity) are a subset.

Figure 4 shows the guts of the circuit programmed as 2-bit **AND** function. The connections of the *f*-terminals to **0** and **1** constitute the instruction word; they are depicted as thick coloured lines in dark red and light blue.

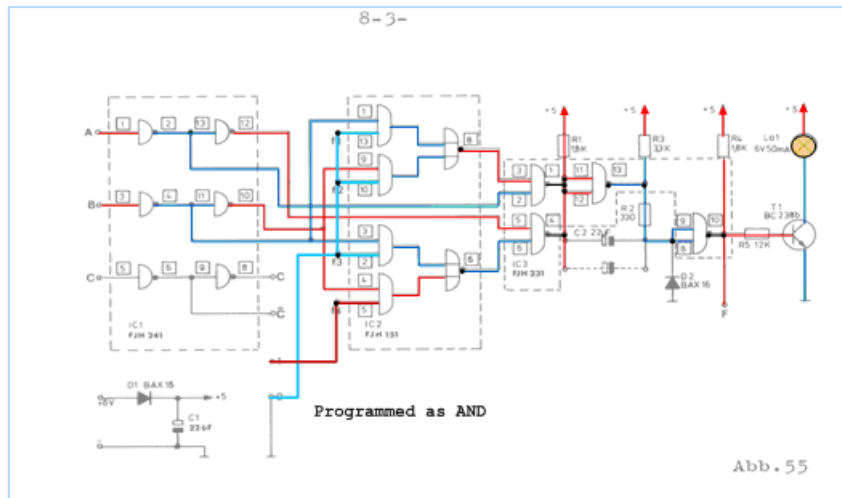


Figure 4

The red and blue colours inside the circuit diagram reflect the voltage levels of the corresponding network node (red **+**, blue **GND**). This should make it easy to follow the various inversions of inputs and intermediate results.

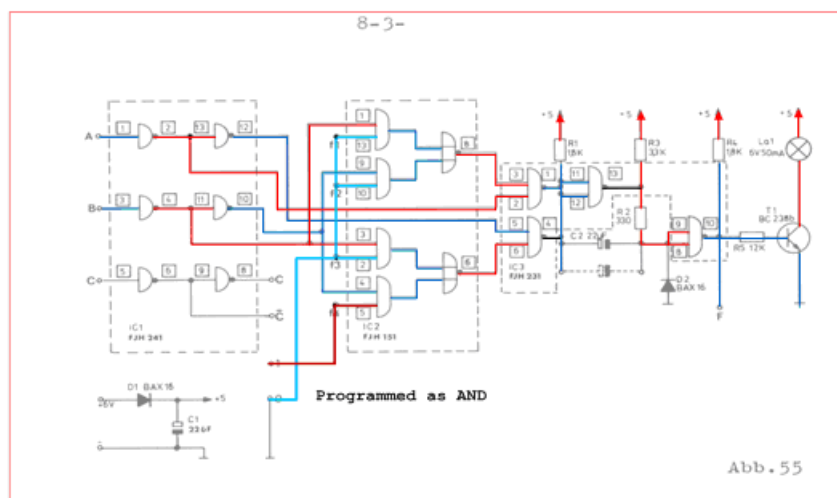


Figure 5

Figure 5 shows the situation where both inputs of the **AND** function are **FALSE (0)**. Using black colour at the output of the **NAND** gates signifies the fact that these are 'open collector' gates – if their logic value is 1, the voltage HIGH level is produced by a pull-up resistor. Notice that the output gate 1 is painted in blue – it pulls down the default **HIGH** level imposed by the resistor. This node effectively implements **OR**-behaviour: if neither of the **NAND** gates is activated, the pull-up is responsible for the output level. If either or both of the **NANDs** are active, the voltage is pulled to **GND**.

Figure 6 shows the circuit programmed as **XOR** function with the node in question being highlighted. In the truth table embedded in the drawing, **X** stands for 'open collector' (which effectively corresponds to **TRUE**). And before you ask: the diode doesn't influence the circuit's logical behaviour. It just cuts off negative spikes from the charged capacitor in situations where its (+) terminal is pulled to **GND**.

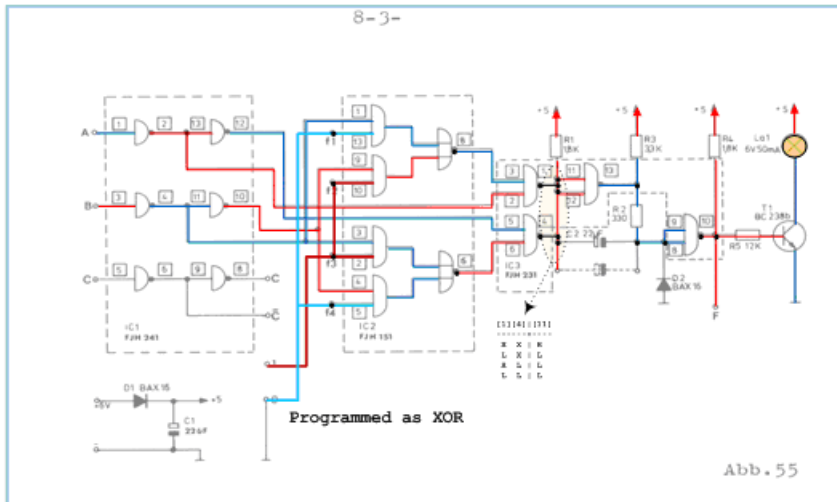


Figure 6

A somewhat remarkable feature of the circuit regards its capability to implement logical functions with *three* variables. The Philips engineers could have applied the template solution used for functions with 2 variables. By doubling the number of f-terminals that can be connected to the *constants* 0 or 1, all  $2^8 = 256$  functions possible with 3 variables can be covered.

Instead, the engineers employed what might be considered *dynamic programming*, a very simple example of data-driven instruction modification on the fly (aka self-modifying code). This is achieved by using the *variable* value of  $c$  and  $\bar{c}$  as part of the “programming instruction” patterns at  $f1$ - $f4$  that determine the functional behaviour of the circuit. In mathematical terms, this works equally well. We now have 4 values (0, 1,  $c$  and  $\bar{c}$ ) to initialize  $f1$ - $f4$ . This gives  $4^4 = 2^8 = 256$  different combinations – exactly the number of distinguishable patterns required to uniquely encode the complete range of 3-variable Boolean functions.

Question: can you construct an equivalent circuit using 2P2T, 3P2T, and/or 4P2T relays? And is there a (provably) minimal solution with regard to number of poles and number of relays? Suggestions are welcome – email [schnoeckus-at-yahoo.com](mailto:schnoeckus-at-yahoo.com)