Elevator control system

by Antonio Bernardini

Table of contents

1 Homework

1.1 Introduction

For this homework you were asked to design and implement the control system of an elevator in a 3-story building, plus the ground floor and a top *VIP floor*, described below. The elevator can be in the following states:

- *Waiting*: The elevator is waiting to be called to a floor.
- *Up or Down Movement*: The elevator is moving in a specific direction.
- *Stop*: The elevator stops upon arrival at a floor where a reservation has been made.
- *Door opening*: The doors open and remain open for 9 seconds.

An arrival sensor is installed on each floor which, when it detects the presence of the elevator, raises a signal A_x to indicate that it has arrived at floor x.

1.2 Floor call management and movement

The passage to a floor can be booked either from the floor itself or from inside the elevator, with the appropriate button panel. It is possible to book the passage to any floor in any state, even while the elevator is moving. The call to a specific floor is buffered in a specifically dedicated memory element and remains in the "request made" state until the elevator arrives at the floor involved in the request.

If already in motion, the elevator proceeds in its direction of travel until it has reached the last floor for which there is a call, in that direction. For example, if the elevator has reached the second floor in upward direction and there are two calls, one for the third and one for the first, the elevator will give priority to the third floor since it can be reached in the same direction of travel.

Conversely, when the elevator reaches a floor and there are only calls from floors reachable in the opposite direction, it will reverse direction.

If there is no call when the doors close, the elevator will remain (with the doors closed) in the waiting state. If in this state the elevator receives a call from the floor where it is present, it will simply open the doors.

This basic operation is modified by the last (fourth) floor: the *VIP floor*. If a request to go to the *VIP floor* has been received, if the elevator is moving in that direction (i.e. upwards), it will *skip all other stops* until it reaches the *VIP floor*. Conversely, if it is moving in the opposite direction (i.e. downwards), it will only reach the next floor for which a request to go has been received and then reverse direction.

1.3 Codings

The elevator shall track the following states, using the following encodings:

z_2	z_1	z_0	State
0	0	0	Waiting
0	0	1	Up Movement
0	1	0	Down Movement
0	1	1	Door Opening
1	0	0	Stop

Table 1. Elevator State Coding

The stop state corresponds to the moment in which the elevator arrives at a floor for which a stop has been requested, signaled by the A_x signal. After stopping, the elevator must open the doors and keep them open for 9 seconds.

The elevator also needs to keep track of which floor it is on at any given moment in time. For this purpose, the following variables are used:

y_2	y_1	y_0	Floor
0	0	0	T
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	VIP

Table 2. Elevator Floor Coding

When the system starts, it can be assumed that the elevator is at floor T in the waiting state.

1.4 Inputs

The elevator control system inputs are of two families: the first family consists of the previously mentioned A_x signals that are raised when an elevator reaches a floor. These signals are only 1 when the elevator is detected, and then return to zero.

The second family consists of the P_x signals that indicate the call to floor x. These signals are worth 1 only when the call button is pressed, and then return to zero. It is therefore necessary to buffer these signals in appropriate ways.

1.5 How to make the circuit

The circuit solution must be inserted into the circuit.dig file, which can be modified using the Digital circuit editor and simulator. The `circuit.dig` file already specifies the inputs and outputs of the circuit.

1.6 Suggestions

1. To measure time, you can use a *clock* set to a suitable frequency and "count" the passage of time with a suitable number of states. For example, to count 5 seconds, you can use a clock set to 1 Hz and use 5 states. The clock in `circuit.dig` is already set to 1 Hz.

2. It is possible to *decompose* a single state machine into multiple state machines that control the operation of different parts of the system in a coordinated manner. The state of one state machine can be used as an input to another. Likewise, the output of one state machine can become the input to another. In this way, it is possible to study the problems separately and compose a final solution.

3. State transitions are not necessarily due to the change of a single input variable. The "input characters" in this case can correspond to a boolean value calculated by an arbitrary boolean function, which processes system input signals and variables stored in appropriate memory elements in an appropriate manner. Some transitions can be "automatic", simply linked to the receipt of a certain number of clock pulses.

2 Solution

2.1 Introduction

Analyzing the `circuit.dig` file for the first time we notice the presence of 10 inputs, respectively $P_0, \dots P_3, P_{\text{VIP}}$ for calls and $A_0, \dots, A_3, A_{\text{VIP}}$ for arrivals, of 6 transitions, respectively y_2, y_1, y_0 for floors and z_2, z_1, z_0 for states, and finally the *clock* set to 1 Hz. Furthermore, there is no explicit output, in fact the transitions, before becoming such, "pass" through D flip-flops by definition of a finite state automaton and, in the case of the elevator, it does not have a real output, but it will be necessary for it to show both the floor and the state at a given time instant, therefore these transitions also act as output. We are faced with a problem that can certainly be modeled with Mealy and/or Moore machines, but not directly. In fact, it is necessary to simplify the problem using the famous *divide and conquer* technique and we will see in this document how to do it. Enjoy the reading.

2.2 Design choices

As for the design choices, I would like to start from the end. In particular, we will implement a system for displaying the floors and elevator states using a 7-segment display for the floors and 4 16-segment displays for the states. In fact, by keeping the outputs of the `circuit.dig` file, we would have a binary encoding of the floors and states and this would make it more difficult to read by an average user who is assumed not to be an engineer. Subsequently, as required by the specifications, we should buffer the P_x calls in specific memory elements. For example, SR flip-flops are more than enough to do this. We will understand how to manage the RESET signals of the SR flip-flops and we will generate several "custom" signals, such as DESTINATION and CK_2, useful for solving small problems. Subsequently, we will try to analyze the problem of request management using specific $\text{UP}, \text{STOP}, \text{DOWN}$ signals to manage priorities through a dedicated circuit, and at the same time we will buffer the previous direction of the elevator, always in specific memory elements. Finally we will deal with the realization of finite state automata, both for states and for planes, with the dedicated circuit for managing the Timer. Let's start!

2.3 7-segment display for floors

It seems absurd to start the design from the end, but it is not. In fact, as previously mentioned, it is not a random choice because in the same way that an average user will have a simplified reading of the floors and the elevator status, even an engineer in the debugging phase can benefit from it. In fact, it is very easy to get the position of 1 bit wrong and read, for example, the floor $y_2 = 0$, $y_1 = 1, y_0 = 0$ (floor 2) instead of the floor $y_2 = 1, y_1 = 0, y_0 = 0$ (*VIP floor*). This also applies to the states of course.

So let's start our design by implementing a 7-segment display for displaying the planes. To do this it is clear that we need to convert a binary input (i.e. y_2, y_1, y_0) into a corresponding decimal number. So we can implement a simple decoder that by definition does just that. Of course we would like to implement the combinational circuit using a ROM (Read Only Memory) to make the design as clean as possible. Having understood this, it is necessary to know that a 7-segment display is nothing more than a set of connected LED diodes that are turned on based on the decoded input value. In general a 7-segment display looks like this:

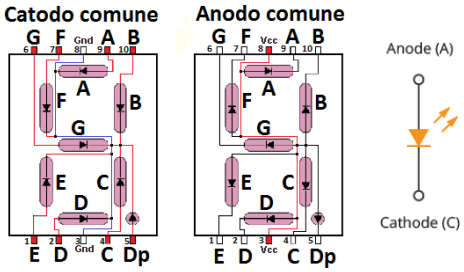


Figure 1. 7-segment display

In particular, we will use the common cathode configuration which, as the name suggests, features LEDs that all share the same cathode connected to GND. As shown in Fig. 1, each input of the 7-segment display will light one or more LEDs, consequently illuminating one or more lines. Therefore, to correctly represent each number, we can refer to the following image:

Figure 2. Encoding numbers for a 7-segment display

For example to write the number 1 we should set $b = 1$ and $c = 1$. Then, as I said before, we should implement a decoder to make sure to activate b, c or any other input using the binary input y_2, y_1, y_0. Therefore we can build the following truth table:

Floor	y_2	y_1	y_0	a	b	c	d	e	f	g	HEX
0	0	0	0	1	1	1	1	1	1	0	0x7E
1	0	0	1	0	1	1	0	0	0	0	0x30
2	0	1	0	1	1	0	1	1	0	1	0x6D
3	0	1	1	1	1	1	1	0	0	1	0x79
PIV	1	0	0	0	1	1	0	0	1	1	0x33

Table 3. Decoder truth table for 7-segment display

At this point we have everything we need to make the decoder, so since we would have 3 input bits and 7 output bits we should set the ROM with the following settings:

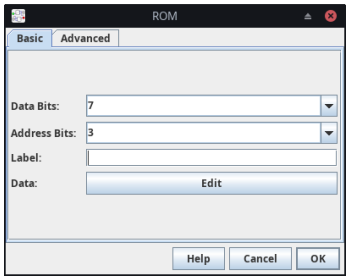


Figure 3. ROM settings for 7-segment display

by inserting the HEX column in Table 3 in the `Data` field. Therefore the final circuit is the following:

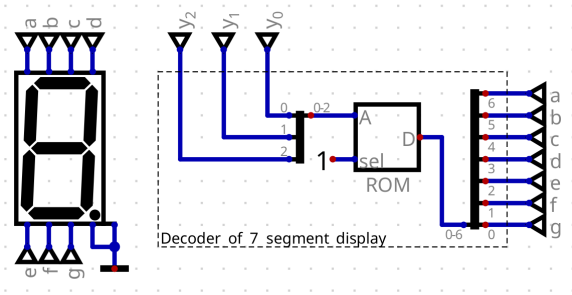


Figure 4. Decoder for 7-segment display

Finally the "dp" input of the 7-segment display handles the dot, but since we don't need it, it has been connected directly to GND.

2.4 16-segment display for status

This design is similar to the previous one although there is an important factor to take into account: the number of bits! In fact since we have to implement 4 16-segment displays we would need $16 \cdot 4 = 64$ bits. However the Digital simulator can create ROMs for a maximum of 32 bits each. Therefore we will implement two ROMs. The first one for the management of the first 32 "less significant" bits and the second one for the management of the last 32 "more significant" bits. But why do we need to use 4 16-segment displays? Wouldn't it be enough to use a 7-segment display again? Well, with a 7-segment display we would only display the numbers from 0 to 4 just like the planes. But in the case of the states we need to give some more information. In fact we can make the following association:

State	D_1	D_2	D_3	D_4
Waiting	W	A	I	T
Up Movement	U	P		
Down Movement	D	O	W	N
Doors Opening	O	P	E	N
Stop	S	T	O	P

Table 4. Conversion of elevator states into characters that can be represented on the 4 16-segment displays

Fortunately, $\frac{3}{4}$ of the states will use exactly all 4 displays and only in the case of the UP state will 2 displays be used to represent the individual characters. Let's continue our discussion keeping in mind that in general a 16-segment display looks like this:

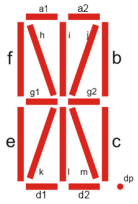


Figure 5. 16 segment display

so for the correct representations of the characters we can refer to the following image:

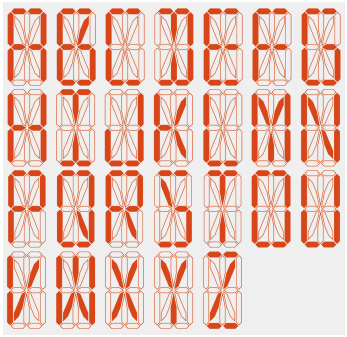


Figure 6. Character encodings for a 16-segment display

We are ready to build the truth table for the first 2 displays i.e. for the first 32 "least significant" bits:

Display D_1																				
State	z_2	z_1	z_0	a_1	a_2	b	c	d_2	d_1	e	f	g_1	g_2	h	i	j	m	l	k	Character	HEX
WAIT	0	0	0	0	0	1	1	0	0	1	1	0	0	0	1	0	1	0	1	W	0xA8CC
UP	0	0	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	U	0x00FC
DOWN	0	1	0	1	1	1	1	1	1	0	0	0	0	0	1	0	0	1	0	D	0x483F
OPEN	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	O	0x00FF
STOP	1	0	0	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	0	S	0x063B
Display D_2																					
State	z_2	z_1	z_0	a_1	a_2	b	c	d_2	d_1	e	f	g_1	g_2	h	i	j	m	l	k	Character	HEX
WAIT	0	0	0	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	A	0x03CF
UP	0	0	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	P	0x03C7
DOWN	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	O	0x00FF
OPEN	0	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	P	0x03C7
STOP	1	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	T	0x4803

Table 5. Decoder truth table for the 16-segment D_1 and D_2 displays

Note that in the conversion from binary to hexadecimal I considered the inverted sequence, i.e. k, l, $m, \ldots$ so that the bit in position 0 is a_1 while the bit in position 15 is k (always considering groups of 4 bits in the conversion from binary to hexadecimal). This is because in the Digital simulator, when using the 16-segment display, the bit in position 0 is precisely a_1, therefore it is necessary to

appropriately encode the values in hexadecimal to be inserted into the ROM. Let's continue with the remaining 2 displays, i.e. for the second 32 "most significant" bits:

Display D_3																			
State	z_2	z_1	z_0	a_1	a_2	b	c	d_2	d_1	e	f	g_1	g_2	h	i	j	m	l	k	Character	HEX
WAIT	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1	0	0	1	0	I	0x4833
UP	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0x0000
DOWN	0	1	0	0	0	1	1	0	0	1	1	0	0	0	1	0	1	0	1	W	0xA8CC
OPEN	0	1	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0	0	E	0x03F3
STOP	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	O	0x00FF
Display D_4																			
State	z_2	z_1	z_0	a_1	a_2	b	c	d_2	d_1	e	f	g_1	g_2	h	i	j	m	l	k	Character	HEX
WAIT	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	T	0x4803
UP	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0x0000
DOWN	0	1	0	0	0	1	1	0	0	1	1	0	0	1	0	0	1	0	0	N	0x24CC
OPEN	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	0	1	0	0	N	0x24CC
STOP	1	0	0	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	P	0x03C7

Table 6. Decoder truth table for the 16-segment D_3 and D_4 displays

At this point we have everything we need to make the decoder, so since we would have 3 bits of input and 32 bits of output we should set the two ROMs with the following settings:

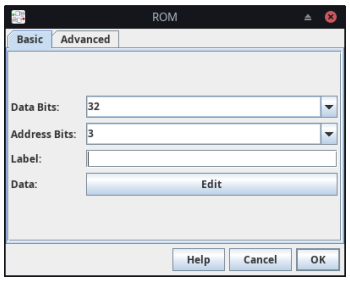
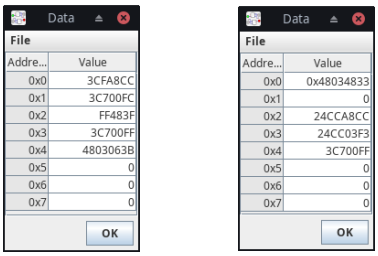


Figure 7. Impostazioni ROM per il display a 16 segmenti

by inserting in the `Data` field the HEX column present in both Table 5 and Table 6 as follows:



D_1 and D_2 ROM D_3 and D_4 ROM

Figure 8. `Data` field of the ROMs

as you can see the hexadecimal values have been arranged in such a way that for display D_1 we complete the first 16 bits of the first ROM and for display D_2 we complete the last 16 bits of the first ROM for a total of 32 bits. The same is true for displays D_3 and D_4 but for the second ROM. Therefore the final circuit is the following:

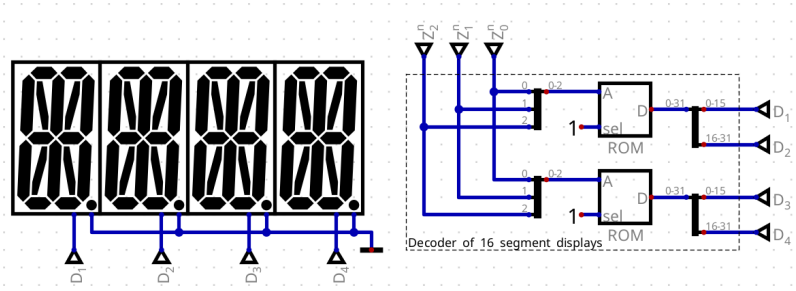


Figure 9. Decoder for 16-segment display

Finally the "dp" input of the 16-segment display handles the dot, but since we don't need it, it has been connected directly to GND.

2.5 Buffering pending requests

Let's get into the heart of the design and in particular we use SR flip-flops to buffer the P_x calls. An SR flip-flop is a circuit that behaves as follows: when the input SET $= 1$ and the input RESET $= 0$ it follows that the output $Q = 1$, while when the input SET $= 0$ and the input RESET $= 1$ it follows that the output $Q = 0$. Finally, the case SET $= 1$ and RESET $= 1$, at the same time, is not admissible. This type of flip-flop is perfect in our case because when a generic P_x call arrives, it transforms into an R_x request (so $Q = R_x$) that remains waiting before being served (i.e. it remains waiting until RESET $= 0$). Therefore we would have all the calls $P_0, \ldots, P_3, P_{\text{VIP}}$ connected to 5 SR flip-flops respectively to the SET inputs, so that, when a generic call arrives, the relative waiting request is activated.

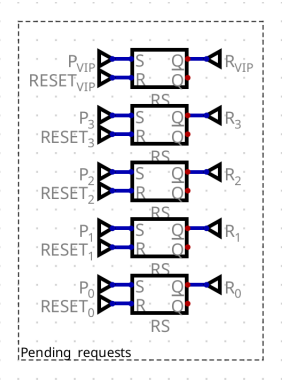


Figure 10. Buffering pending requests

2.6 Reset Management

But who do we connect to the RESET inputs? We might think of connecting the arrivals $A_0, \ldots, A_3$, A_{VIP}, but this is not good because the arrivals behave like sensors for detecting the passage through a generic floor but are not indicators of whether a generic request R_x, and therefore a generic call P_x, has been served or not. So we need to implement a circuitry that tells us if the request has been served correctly so that we can proceed with the actual reset of the SR flip-flops involved. But when do we have to reset the SR flip-flops? Or rather, when is a request served? Well, a request is served if we have arrived at a floor x and if 9 seconds have passed in the door open state. Therefore it is clear that the bits y_2, y_1, y_0, which represent a generic floor, must be ANDed (with appropriate negated inputs) between them and we must also add a TIMEOUT signal which represents the overflow bit when 9 seconds are counted. In particular, using a Timer and setting a number of bits equal to 4

and as a maximum counting value $(8)_{10} = (1000)_2$ when we move to $(9)_{10} = (1001)_2$ we have one bit left to generate the overflow which will be assigned to the TIMEOUT signal, but we will discuss this better later when the finite state automaton for managing the elevator states is created.

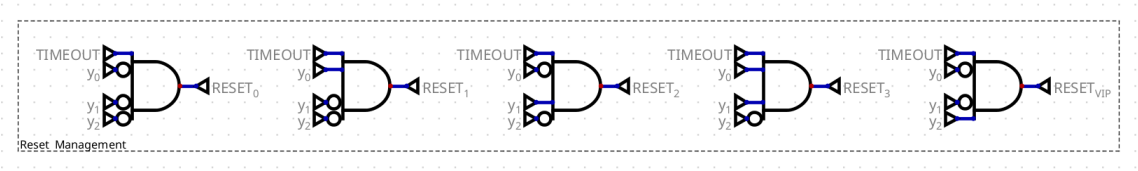


Figure 11. Reset Management

2.7 Detector for a generic floor

In general we know that the arrival signals $A_0, \ldots, A_3, A_{\text{VIP}}$ differ from the calls $P_0, \ldots, P_3, P_{\text{VIP}}$ because they are unique. In fact, there cannot be two arrival signals active at the same time. Therefore we can build a detector that activates only when passing through a generic floor and to do so it is sufficient to OR all the arrival signals $A_0, \ldots, A_3, A_{\text{VIP}}$ precisely because of their definition of uniqueness.

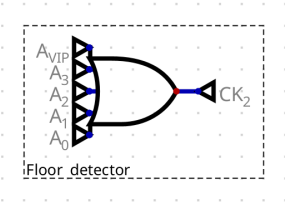


Figure 12. Detector for a generic floor

2.8 Signal for arrival at destination

Another useful signal to implement is DESTINATION, a signal that is 1 when the elevator arrives at its destination, otherwise it is 0. To implement it we should satisfy a generic R_x request in conjunction with a generic arrival signal A_x. This translates into 5 AND gates plus the addition of an OR gate (it is basically a Multiplexer, but by individually implementing the 6 total gates we save inputs that would not be used, in fact 5 is not a power of 2). In fact, precisely because of the uniqueness of the A_x signals, only one of the AND gates will be activated in rotation. Naturally, arriving at the destination does not imply that an R_x request has been served.

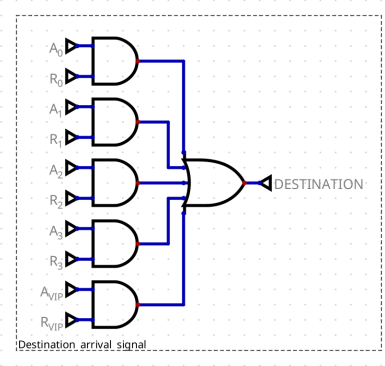


Figure 13. Signal for arrival at destination

2.9 Request Management

At this point we need to find a way to "know" if the elevator should go up, down or stay still. Therefore we certainly know that there will have to be 3 signals, namely UP, STOP and DOWN, which, based on a generic request R_x and knowing which floor the elevator has arrived at, will be activated appropriately. But how do we know which floor the elevator has arrived at? Well, simply by using the arrival signals $A_0, \ldots, A_3, A_{\text{VIP}}$ (be careful not to confuse this logic with the logic of the DESTINATION signal). But how? Considering, for example, the UP signal we know that the elevator must go up only if $\exists A_x > x \wedge R_x = 1$ (the symbol $\wedge$ represents the logical AND), that is, only if I receive a request R_x and the elevator arrives at a floor x with a signal $A_x > x$. For example, if $x = 0$ (floor 0) and I receive a request $R_1 = 1$ to know if the elevator has arrived at floor 1, we would have $A_1 = 1$ and therefore $A_1 > x$. So:

$$\begin{aligned} \text{UP} = 1 &\iff \exists A_x > x \wedge R_x = 1 \\ \text{STOP} = 1 &\iff \exists A_x = x \wedge R_x = 1 \\ \text{DOWN} = 1 &\iff \exists A_x < x \wedge R_x = 1 \end{aligned}$$

Therefore, the only iterative logic network that allows us to perform the comparisons $A_x > x$, $A_x = x$, $A_x < x$ is the comparator that will have the UP, STOP, DOWN signals as outputs. This comparator will have to manage all the possible R_x requests based on the arrival floors A_x for a total of $2^5 \cdot 5 = 160$ cases (in fact we have 4 floors plus the ground floor). Furthermore, the arrival signals A_x will have to be buffered, based on the detection of a generic floor, to know which signal to activate between UP, STOP, DOWN based on the arrival of a new R_x request. And here is where the circuit in Fig. 12 comes to our aid. In particular, using a D flip-flop with a CK_2 *clock* signal, we can make the D input pass to the Q output (i.e. $Q = D$) only when a new floor is detected. Furthermore, to make our life easier, we can encode the A_x signals by going from 5 bits to 3 bits. This last choice is very useful for the construction of the comparator truth tables to ensure that there are 8 bits in input (i.e. the 5 requests R_x plus 3 bits of the coding of the arrival signals A_x) and 3 bits in output (i.e. UP, STOP, DOWN).

Let's start by realizing the encoder of the arrival signals A_x, which will have the following truth table:

A_{VIP}	A_3	A_2	A_1	A_0	a_2	a_1	a_0	HEX
0	0	0	0	0	0	0	0	0x0
0	0	0	0	1	0	0	0	0x0
0	0	0	1	0	0	0	1	0x1
0	0	1	0	0	0	1	0	0x2
0	1	0	0	0	0	1	1	0x3
1	0	0	0	0	1	0	0	0x4

Table 7. Encoder truth table for arrival signals

To implement the encoder, a ROM (Read Only Memory) will be used, therefore the following settings must be set:

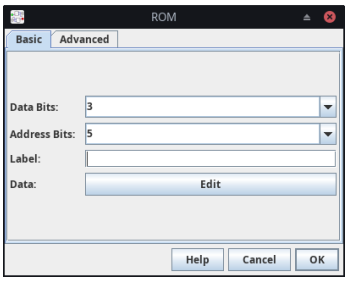


Figure 14. ROM settings for the encoder

by inserting the HEX column in Table 7 in the `Data` field. Subsequently, all the output bits will be stored in a D flip-flop, with *clock* signal CK_2, with the following settings:

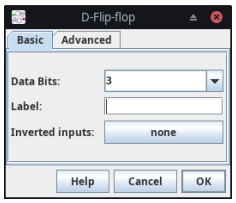


Figure 15. D Flip-Flop Settings

and the Q output, having 3 bits, will be used as the comparator input together with the R_x requests. At the end of this part the circuit is the following:

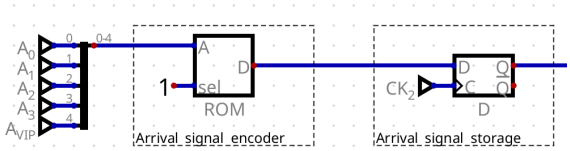


Figure 16. Incomplete request management

Now we report the relationships that link the output signals UP, STOP, DOWN, with the relative inputs in 5 truth tables each composed of 32 bits. The first truth table represents the 32 possible requests R_x in the case in which $a_2 = 0, a_1 = 0, a_0 = 0$ (so we are in the case in which no arrival signal is activated, or the arrival signal A_0 is activated):

a_2	a_1	a_0	R_0	R_1	R_2	R_3	R_{VIP}	UP	STOP	DOWN	HEX
0	0	0	0	0	0	0	0	0	0	0	0x0
0	0	0	0	0	0	0	1	1	0	0	0x4
0	0	0	0	0	0	1	0	1	0	0	0x4
0	0	0	0	0	0	1	1	1	0	0	0x4
0	0	0	0	0	1	0	0	1	0	0	0x4
0	0	0	0	0	1	0	1	1	0	0	0x4
0	0	0	0	0	1	1	0	1	0	0	0x4
0	0	0	0	0	1	1	1	1	0	0	0x4
0	0	0	0	1	0	0	0	1	0	0	0x4
0	0	0	0	1	0	0	1	1	0	0	0x4
0	0	0	0	1	0	1	0	1	0	0	0x4
0	0	0	0	1	0	1	1	1	0	0	0x4
0	0	0	0	1	1	0	0	1	0	0	0x4
0	0	0	0	1	1	0	1	1	0	0	0x4
0	0	0	0	1	1	1	0	1	0	0	0x4
0	0	0	0	1	1	1	1	1	0	0	0x4
0	0	0	1	0	0	0	0	0	1	0	0x2
0	0	0	1	0	0	0	1	1	1	0	0x6
0	0	0	1	0	0	1	0	1	1	0	0x6
0	0	0	1	0	0	1	1	1	1	0	0x6
0	0	0	1	0	1	0	0	1	1	0	0x6
0	0	0	1	0	1	0	1	1	1	0	0x6
0	0	0	1	0	1	1	0	1	1	0	0x6
0	0	0	1	0	1	1	1	1	1	0	0x6
0	0	0	1	1	0	0	0	1	1	0	0x6
0	0	0	1	1	0	0	1	1	1	0	0x6
0	0	0	1	1	0	1	0	1	1	0	0x6
0	0	0	1	1	0	1	1	1	1	0	0x6
0	0	0	1	1	1	0	0	1	1	0	0x6
0	0	0	1	1	1	0	1	1	1	0	0x6
0	0	0	1	1	1	1	0	1	1	0	0x6
0	0	0	1	1	1	1	1	1	1	0	0x6

Table 8. Comparator truth table for $a_2 = 0, a_1 = 0, a_0 = 0$

The second truth table represents the 32 possible R_x requests in the case where $a_2 = 0, a_1 = 0, a_0 = 1$

(so we are in the case where the arrival signal A_1 is activated):

a_2	a_1	a_0	R_0	R_1	R_2	R_3	R_{VIP}	UP	STOP	DOWN	HEX
0	0	1	0	0	0	0	0	0	0	0	0x0
0	0	1	0	0	0	0	1	1	0	0	0x4
0	0	1	0	0	0	1	0	1	0	0	0x4
0	0	1	0	0	0	1	1	1	0	0	0x4
0	0	1	0	0	1	0	0	1	0	0	0x4
0	0	1	0	0	1	0	1	1	0	0	0x4
0	0	1	0	0	1	1	0	1	0	0	0x4
0	0	1	0	0	1	1	1	1	0	0	0x4
0	0	1	0	1	0	0	0	0	1	0	0x2
0	0	1	0	1	0	0	1	1	1	0	0x6
0	0	1	0	1	0	1	0	1	1	0	0x6
0	0	1	0	1	0	1	1	1	1	0	0x6
0	0	1	0	1	1	0	0	1	1	0	0x6
0	0	1	0	1	1	0	1	1	1	0	0x6
0	0	1	0	1	1	1	0	1	1	0	0x6
0	0	1	0	1	1	1	1	1	1	0	0x6
0	0	1	1	0	0	0	0	0	0	1	0x1
0	0	1	1	0	0	0	1	1	0	1	0x5
0	0	1	1	0	0	1	0	1	0	1	0x5
0	0	1	1	0	0	1	1	1	0	1	0x5
0	0	1	1	0	1	0	0	1	0	1	0x5
0	0	1	1	0	1	0	1	1	0	1	0x5
0	0	1	1	0	1	1	0	1	0	1	0x5
0	0	1	1	0	1	1	1	1	0	1	0x5
0	0	1	1	1	0	0	0	0	1	1	0x3
0	0	1	1	1	0	0	1	1	1	1	0x7
0	0	1	1	1	0	1	0	1	1	1	0x7
0	0	1	1	1	0	1	1	1	1	1	0x7
0	0	1	1	1	1	0	0	1	1	1	0x7
0	0	1	1	1	1	0	1	1	1	1	0x7
0	0	1	1	1	1	1	0	1	1	1	0x7
0	0	1	1	1	1	1	1	1	1	1	0x7

Table 9. Comparator truth table for $a_2 = 0, a_1 = 0, a_0 = 1$

The third truth table represents the 32 possible R_x requests in the case where $a_2 = 0, a_1 = 1, a_0 = 0$ (so we are in the case where the arrival signal A_2 is activated):

a_2	a_1	a_0	R_0	R_1	R_2	R_3	R_{VIP}	UP	STOP	DOWN	HEX
0	1	0	0	0	0	0	0	0	0	0	0x0
0	1	0	0	0	0	0	1	1	0	0	0x4
0	1	0	0	0	0	1	0	1	0	0	0x4
0	1	0	0	0	0	1	1	1	0	0	0x4
0	1	0	0	0	1	0	0	0	1	0	0x2
0	1	0	0	0	1	0	1	1	1	0	0x6
0	1	0	0	0	1	1	0	1	1	0	0x6
0	1	0	0	0	1	1	1	1	1	0	0x6
0	1	0	0	1	0	0	0	0	0	1	0x1
0	1	0	0	1	0	0	1	1	0	1	0x5
0	1	0	0	1	0	1	0	1	0	1	0x5
0	1	0	0	1	0	1	1	1	0	1	0x5
0	1	0	0	1	1	0	0	0	1	1	0x3
0	1	0	0	1	1	0	1	1	1	1	0x7
0	1	0	0	1	1	1	0	1	1	1	0x7
0	1	0	0	1	1	1	1	1	1	1	0x7
0	1	0	1	0	0	0	0	0	0	1	0x1
0	1	0	1	0	0	0	1	1	0	1	0x5
0	1	0	1	0	0	1	0	1	0	1	0x5
0	1	0	1	0	0	1	1	1	0	1	0x5
0	1	0	1	0	1	0	0	0	1	1	0x3
0	1	0	1	0	1	0	1	1	1	1	0x7
0	1	0	1	0	1	1	0	1	1	1	0x7
0	1	0	1	0	1	1	1	1	1	1	0x7
0	1	0	1	1	0	0	0	0	0	1	0x1
0	1	0	1	1	0	0	1	1	0	1	0x5
0	1	0	1	1	0	1	0	1	0	1	0x5
0	1	0	1	1	0	1	1	1	0	1	0x5
0	1	0	1	1	1	0	0	0	1	1	0x3
0	1	0	1	1	1	0	1	1	1	1	0x7
0	1	0	1	1	1	1	0	1	1	1	0x7
0	1	0	1	1	1	1	1	1	1	1	0x7

Table 10. Comparator truth table for $a_2 = 0, a_1 = 1, a_0 = 0$

The fourth truth table represents the 32 possible requests R_x in the case where $a_2 = 0, a_1 = 1, a_0 = 1$

(so we are in the case where the arrival signal A_3 is activated):

a_2	a_1	a_0	R_0	R_1	R_2	R_3	R_{VIP}	UP	STOP	DOWN	HEX
0	1	1	0	0	0	0	0	0	0	0	0x0
0	1	1	0	0	0	0	1	1	0	0	0x4
0	1	1	0	0	0	1	0	0	1	0	0x2
0	1	1	0	0	0	1	1	1	1	0	0x6
0	1	1	0	0	1	0	0	0	0	1	0x1
0	1	1	0	0	1	0	1	1	0	1	0x5
0	1	1	0	0	1	1	0	0	1	1	0x3
0	1	1	0	0	1	1	1	1	1	1	0x7
0	1	1	0	1	0	0	0	0	0	1	0x1
0	1	1	0	1	0	0	1	1	0	1	0x5
0	1	1	0	1	0	1	0	0	1	1	0x3
0	1	1	0	1	0	1	1	1	1	1	0x7
0	1	1	0	1	1	0	0	0	0	1	0x1
0	1	1	0	1	1	0	1	1	0	1	0x5
0	1	1	0	1	1	1	0	0	1	1	0x3
0	1	1	0	1	1	1	1	1	1	1	0x7
0	1	1	1	0	0	0	0	0	0	1	0x1
0	1	1	1	0	0	0	1	1	0	1	0x5
0	1	1	1	0	0	1	0	0	1	1	0x3
0	1	1	1	0	0	1	1	1	1	1	0x7
0	1	1	1	0	1	0	0	0	0	1	0x1
0	1	1	1	0	1	0	1	1	0	1	0x5
0	1	1	1	0	1	1	0	0	1	1	0x3
0	1	1	1	0	1	1	1	1	1	1	0x7
0	1	1	1	1	0	0	0	0	0	1	0x1
0	1	1	1	1	0	0	1	1	0	1	0x5
0	1	1	1	1	0	1	0	0	1	1	0x3
0	1	1	1	1	0	1	1	1	1	1	0x7
0	1	1	1	1	1	0	0	0	0	1	0x1
0	1	1	1	1	1	0	1	1	0	1	0x5
0	1	1	1	1	1	1	0	0	1	1	0x3
0	1	1	1	1	1	1	1	1	1	1	0x7

Table 11. Comparator truth table for $a_2 = 0, a_1 = 1, a_0 = 1$

The fifth truth table represents the 32 possible R_x requests in the case where $a_2 = 1, a_1 = 0, a_0 = 0$ (so we are in the case where the arrival signal A_{VIP} is activated):

a_2	a_1	a_0	R_0	R_1	R_2	R_3	R_{VIP}	UP	STOP	DOWN	HEX
1	0	0	0	0	0	0	0	0	0	0	0x0
1	0	0	0	0	0	0	1	0	1	0	0x2
1	0	0	0	0	0	1	0	0	0	1	0x1
1	0	0	0	0	0	1	1	0	1	1	0x3
1	0	0	0	0	1	0	0	0	0	1	0x1
1	0	0	0	0	1	0	1	0	1	1	0x3
1	0	0	0	0	1	1	0	0	0	1	0x1
1	0	0	0	0	1	1	1	0	1	1	0x3
1	0	0	0	1	0	0	0	0	0	1	0x1
1	0	0	0	1	0	0	1	0	1	1	0x3
1	0	0	0	1	0	1	0	0	0	1	0x1
1	0	0	0	1	0	1	1	0	1	1	0x3
1	0	0	0	1	1	0	0	0	0	1	0x1
1	0	0	0	1	1	0	1	0	1	1	0x3
1	0	0	0	1	1	1	0	0	0	1	0x1
1	0	0	0	1	1	1	1	0	1	1	0x3
1	0	0	1	0	0	0	0	0	0	1	0x1
1	0	0	1	0	0	0	1	0	1	1	0x3
1	0	0	1	0	0	1	0	0	0	1	0x1
1	0	0	1	0	0	1	1	0	1	1	0x3
1	0	0	1	0	1	0	0	0	0	1	0x1
1	0	0	1	0	1	0	1	0	1	1	0x3
1	0	0	1	0	1	1	0	0	0	1	0x1
1	0	0	1	0	1	1	1	0	1	1	0x3
1	0	0	1	1	0	0	0	0	0	1	0x1
1	0	0	1	1	0	0	1	0	1	1	0x3
1	0	0	1	1	0	1	0	0	0	1	0x1
1	0	0	1	1	0	1	1	0	1	1	0x3
1	0	0	1	1	1	0	0	0	0	1	0x1
1	0	0	1	1	1	0	1	0	1	1	0x3
1	0	0	1	1	1	1	0	0	0	1	0x1
1	0	0	1	1	1	1	1	0	1	1	0x3

Table 12. Comparator truth table for $a_2 = 1, a_1 = 0, a_0 = 0$

To implement the comparator a ROM (Read Only Memory) will be used, therefore the following settings must be set:

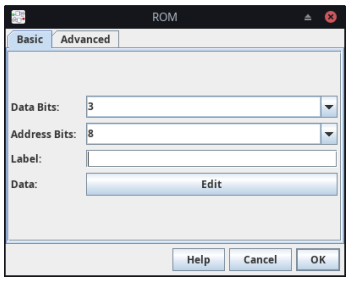


Figure 17. ROM settings for comparator

by inserting the HEX column of the previous 5 tables in the `Data` field. Therefore the final circuit is the following:

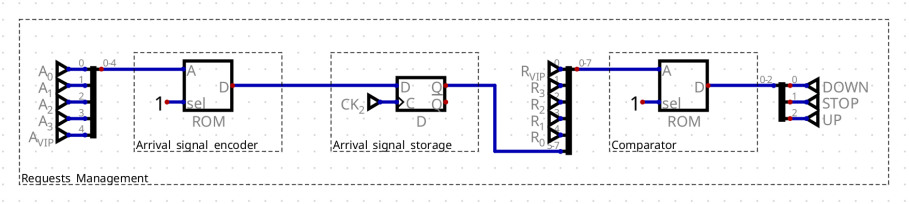


Figure 18. Request Management

2.10 Direction buffering

At this point the issue is a bit delicate. We need to find a way to memorize (and this already gives us an important indication) from which direction the elevator arrives at a generic floor x. That is, whether it arrived from the upper floors or from the lower floors. Or rather, whether once it arrived at a floor x it was at a floor $x+i$ or $x-i$, with $i \in [0,4]$ (where the upper limit 4 would be the *VIP floor*). So we need a signal that must be activated and memorized when the direction is downwards (the upward case manifests itself by duality because if the elevator does not have the downward direction it must necessarily have it upwards), to indicate to the arrival floor that we arrived there by going downwards. This signal, which we will indicate with WAS_{DOWN}, can be memorized in an SR flip-flop, but who sets and/or resets it? We will need two other dedicated signals. The signal that will take care of setting $\text{WAS}_{\text{DOWN}} = 1$ will have to be DIR_{DOWN} which will depend on the elevator state finite automaton, so we will talk about it later. While the signal that will take care of setting $\text{WAS}_{\text{DOWN}} = 0$ (that is, resetting it) is precisely the DOWN signal that we created in Fig. 18, which is always set to 0 except in the case where an R_x request occurs downwards. This means that the SR flip-flop must have the RESET input negated.

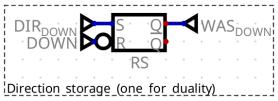


Figure 19. Direction buffering

2.11 Finite State Automaton for State Management

Once all the signals we have discussed so far have been created, it is clear that they need to be inserted as inputs for the finite state automaton dedicated to managing the elevator states. So, to recap, we would have the input signals $\text{CK}_2, \text{DESTINATION}, \text{UP}, \text{STOP}, \text{DOWN}, \text{WAS}_{\text{DOWN}}$ plus the TIMEOUT signal, which we have not yet covered in depth, and finally the R_{VIP} request signal for the management dedicated to the *VIP floor*. Naturally, the transitions z_2, z_1, z_0 are added to these, which are inputs by definition. However, even if the elevator does not have an explicit output, we can still make the finite state automaton generate outputs for us. In particular, in the

circuit in Fig. 19 we talked about the DIR_{DOWN} signal. This signal will be an output of the finite state automaton together with the DIR_{UP} signal, which will be useful for creating the finite state automaton for managing the elevator floors. So, as you can see, it is clear that the states of the finite state automaton for managing the elevator states have as states the encoding found in Table 1. Therefore, we can construct a relationship between these states using the following finite state automaton:

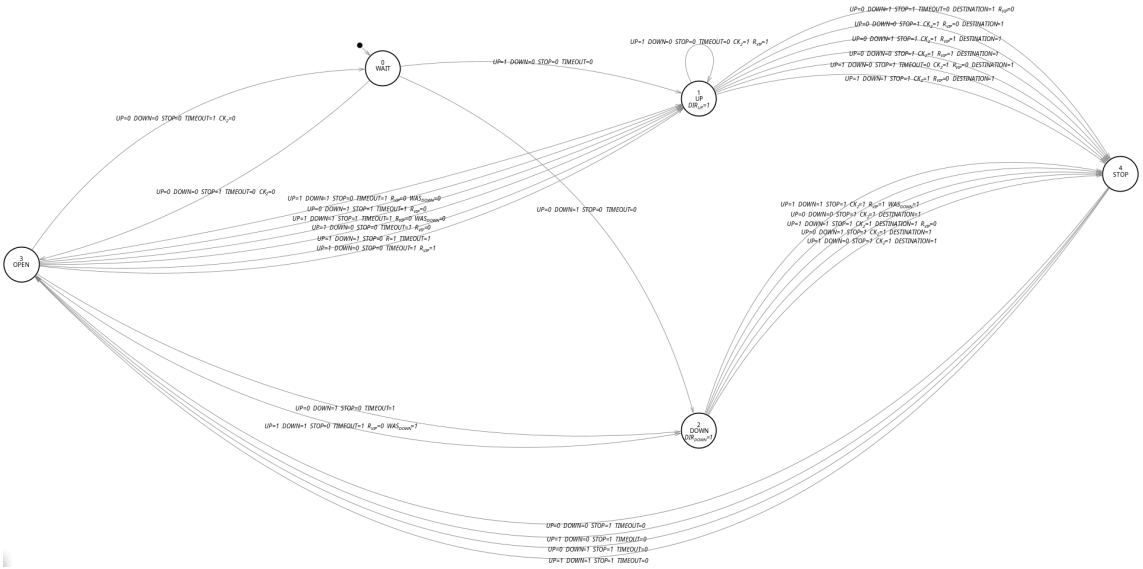


Figure 20. Finite State Automaton for State Management

consultable directly from the `FSM-states.fsm` file through the Digital simulator. Therefore, by going to the appropriate menu `Create > State Transition Table` and then, always from the appropriate menu, on `Create > Circuit Variants > Circuit with LUTs` you get the following sequential circuit:

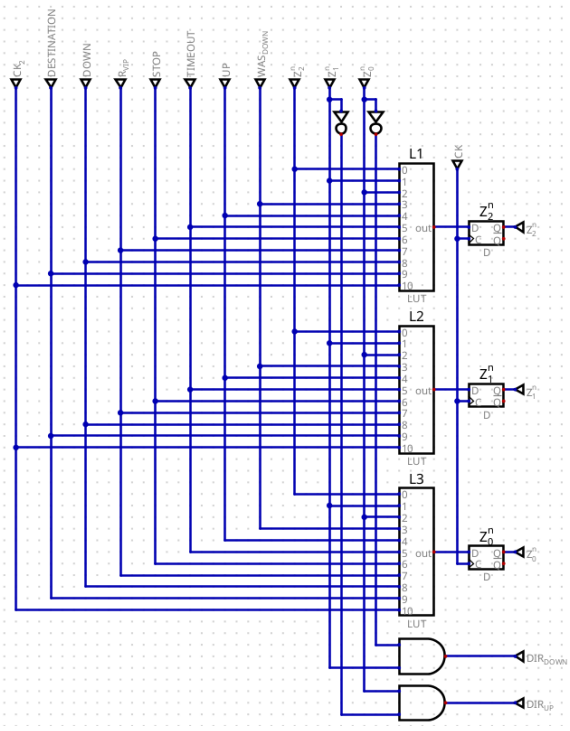


Figure 21. (Incomplete) sequential circuit of the finite state automaton for state management

The circuit shown in Fig. 21 has been slightly modified from the one automatically generated by Digital. In particular, the input and output buttons have been removed, because we use the *tunnel* component to carry a certain data from one part of the circuit to another.

2.11.1 7-segment display for timer

Here too we will implement a decoder to display the Timer count on a 7-segment common cathode display. The design is very similar to the one previously created for the display of the current plane, however, in this case, once again, the number of bits to be decoded changes. Therefore we can construct the following truth table (note that starting from 0 and arriving at 8, 9 seconds have actually passed):

Seconds	t_3	t_2	t_1	t_0	a_t	b_t	c_t	d_t	e_t	f_t	g_t	HEX
0	0	0	0	0	1	1	1	1	1	1	0	0x7E
1	0	0	0	1	0	1	1	0	0	0	0	0x30
2	0	0	1	0	1	1	0	1	1	0	1	0x6D
3	0	0	1	1	1	1	1	1	0	0	1	0x79
4	0	1	0	0	0	1	1	0	0	1	1	0x33
5	0	1	0	1	1	0	1	1	0	1	1	0x5B
6	0	1	1	0	1	0	1	1	1	1	1	0x5F
7	0	1	1	1	1	1	1	0	0	0	0	0x70
8	1	0	0	0	1	1	1	1	1	1	1	0x7F

Table 13. Decoder truth table for 7-segment display

At this point we have everything we need to make the decoder, so since we would have 4 input bits and 7 output bits we should set the ROM with the following settings:

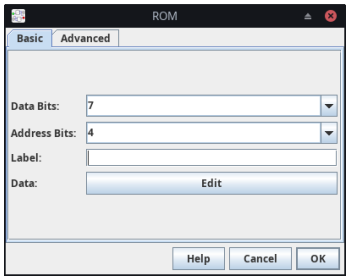


Figure 22. ROM settings for 7-segment display

by inserting the HEX column in Table 13 in the `Data` field. Therefore the final circuit is the following:

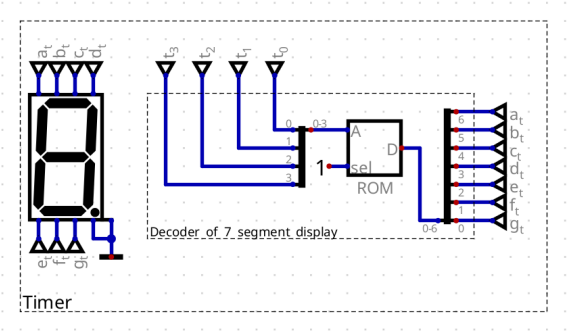


Figure 23. Decoder for 7-segment display

Finally the "dp" input of the 7-segment display handles the dot, but since we don't need it, it has been connected directly to GND.

2.11.2 Timer Management

But when should the Timer be activated? Well, only in the door opening state (i.e. OPEN) which corresponds to $z_2 = 0, z_1 = 1, z_0 = 1$. So we can create the following circuit which connects directly to the previous one:

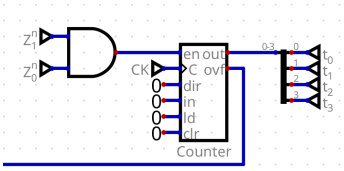


Figure 24. Timer management in finite state automaton for elevator state management

Therefore the complete sequential circuit for managing the elevator states is as follows:

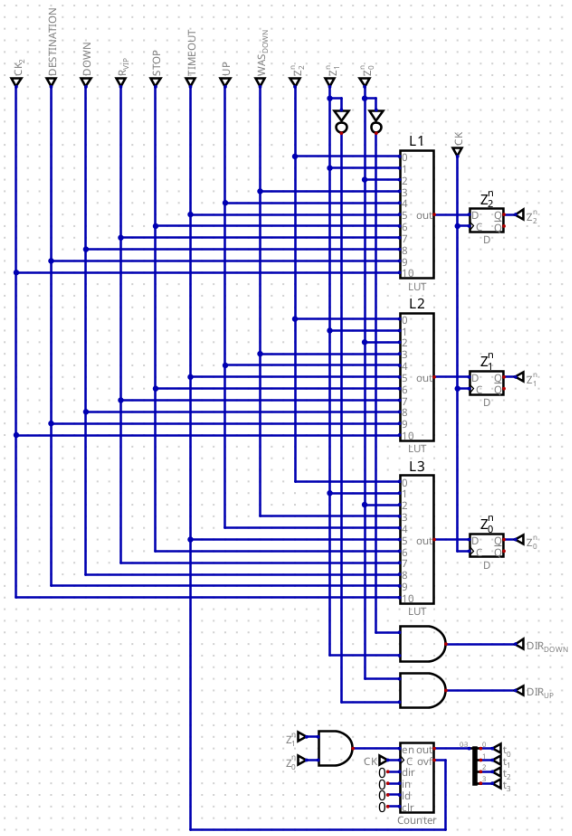


Figure 25. Complete sequential circuit of the finite state automaton for state management

by appropriately connecting the Timer overflow output to the TIMEOUT signal. So in case of overflow, that is when the Timer reaches $(9)_{10}$, we will have TIMEOUT $= 1$.

2.12 Finite State Automaton for Floor Management

We are almost at the end of our design. All that remains is the creation of the finite state automaton for the management of the elevator floors. It will have the CK_2 signal as input together with the outputs generated by the finite state automaton for the management of the elevator states, namely

$\mathrm{DIR_{DOWN}}$ and $\mathrm{DIR_{UP}}$. Naturally, the transitions y_2, y_1, y_0 are added to these, which are inputs by definition. Therefore, as you can imagine, it is clear that the states of the finite state automaton for the management of the elevator floors have as states the encoding found in Table 2. Therefore, we can build a relationship between these states using the following finite state automaton:

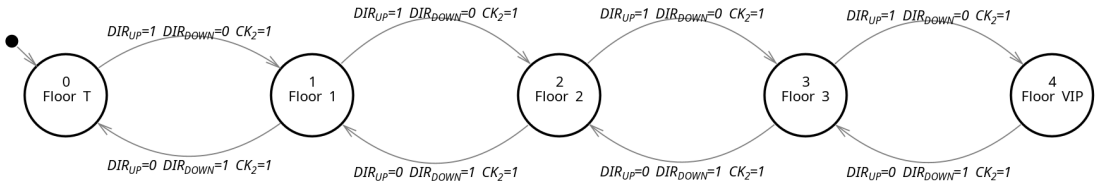


Figure 26. Finite State Automaton for Plan Management

consultable directly from the `FSM-floor.fsm` file through the Digital simulator. Therefore, by going to the appropriate menu `Create > State Transition Table` (I recommend changing the variables from z_* to y_* with a right mouse click on the name) and subsequently, always from the appropriate menu, on `Create > Circuit Variants > Circuit with LUTs` you obtain the following sequential circuit:

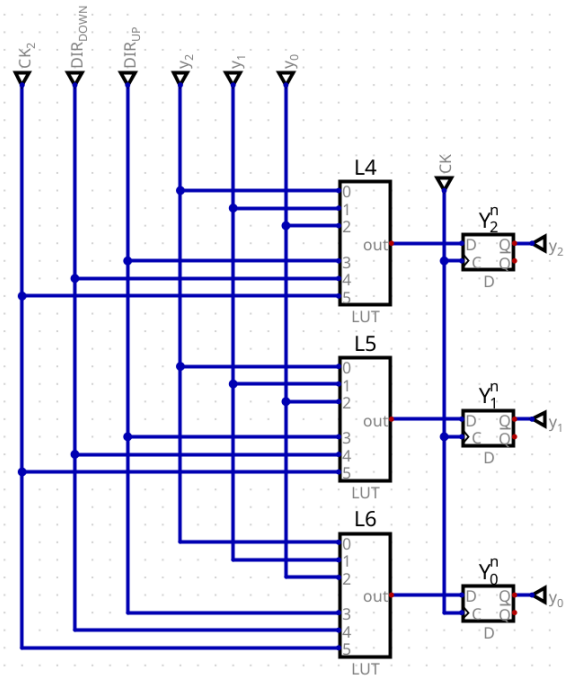


Figure 27. Complete sequential circuit of finite state automaton for plan management

Also in this case the circuit shown in Fig. 27 has been slightly modified compared to the one automatically generated by Digital. In particular the input and output buttons have been removed, because we use the *tunnel* component to carry a certain data from one part of the circuit to another.

2.13 Conclusion

First of all I conclude by showing the complete final circuit that realizes the control system for the elevator:

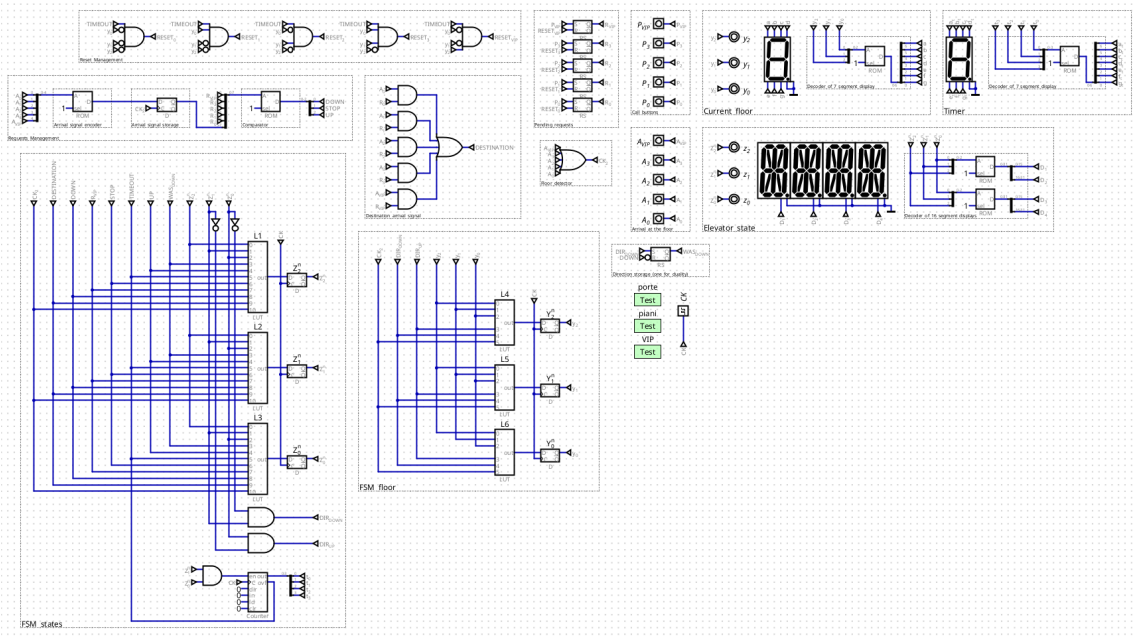


Figure 28. Complete circuit for the management of an elevator control system

consultable directly from the `circuit.dig` file through the Digital simulator. Finally I want to point out that such a design nowadays is very "old school" in the sense that probably, both for a question of costs and for a question of time, it is easier to use programmable embedded boards. However I admit that it is still an exercise in thinking outside the canonical schemes, exactly what an engineer must be able to do. It would be interesting to carry on the design trying to optimize the number of gates to a minimum and then prototype the entire project on PCB using software like KiCAD, adding real sensors, in place of the $A_0, \dots A_3, A_{\text{VIP}}$ signals, motors, and in general everything needed to build an elevator.