Safe Manager

~Documentation~

Szabó Loránd

Description

The ‘Safe Manager’ project is an automaton that has the goal of managing safes or cabinets closed and encoded with 3-digit codes given by the user.

First of all, it engages particularly 9 safes, indexed from 1 to 9, and the digits of the code can have values from 0 to F in hexadecimal. The user, firstly, has to enter the address (index) of the safe, after which the 3-digit code. Both operations are possible due to the *up*, *down* and *add_digit* buttons.

If a safe is open and if the user entered the data, then the safe will be closed and its password/ code will be set to the user-given code.

On the other hand, if a safe is closed and the corresponding code was entered, then it will open. Otherwise, it will stay closed.

Users’ perspective

Inputs:

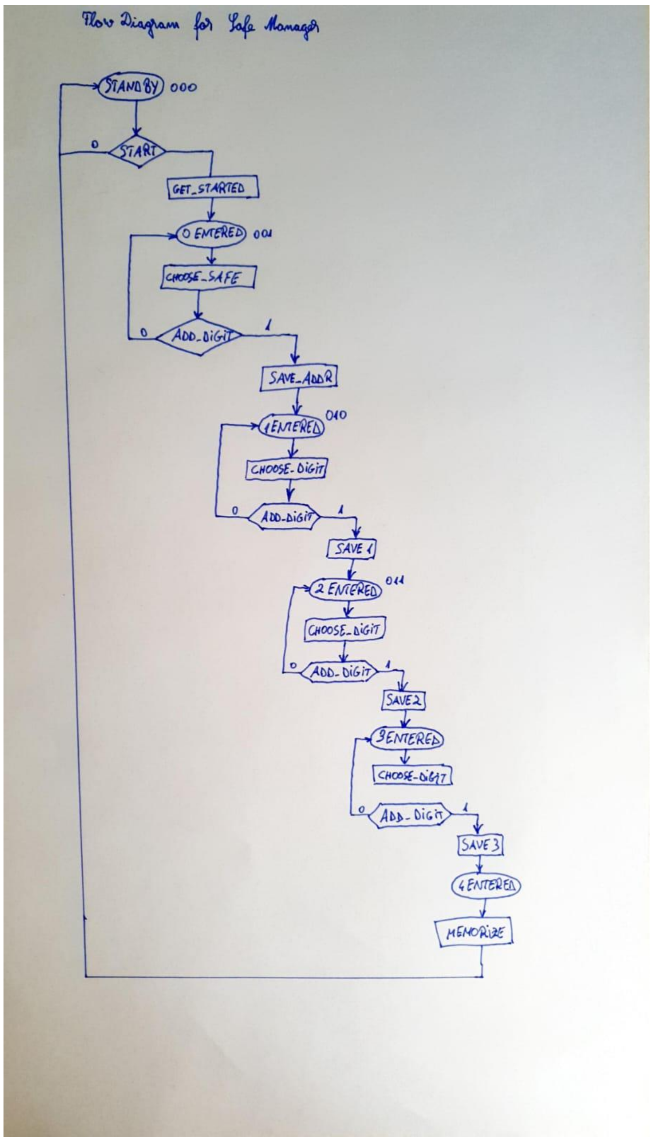
- *Start button:* must be pressed before the user starts to enter the address of the state
- *Add_digit button:* confirms the digit chosen by the user and adds it to the code, also, must be pressed after the sequence of 4 digits is selected (address + code)
- *Up/ Down buttons:* are used to navigate between digits
- *Reset:* resets every code to “000” and opens every safe, thus, it must not be reachable by the user
    - *Remark:* when starting the automata, it must be reset

Outputs:

- *Status LEDs:* there are 9 of them, representing whether a safe is closed (on) or open (off)
- *Seven Segment Display:* show the entered digits: the left most represents the address and the other 3, the code

Implementation

The flow diagram of the automata is the following:



As can be seen, it is quite linear, so its state functions can be easily implemented using a counter. According to the diagram, parallel load is necessary in the following cases:

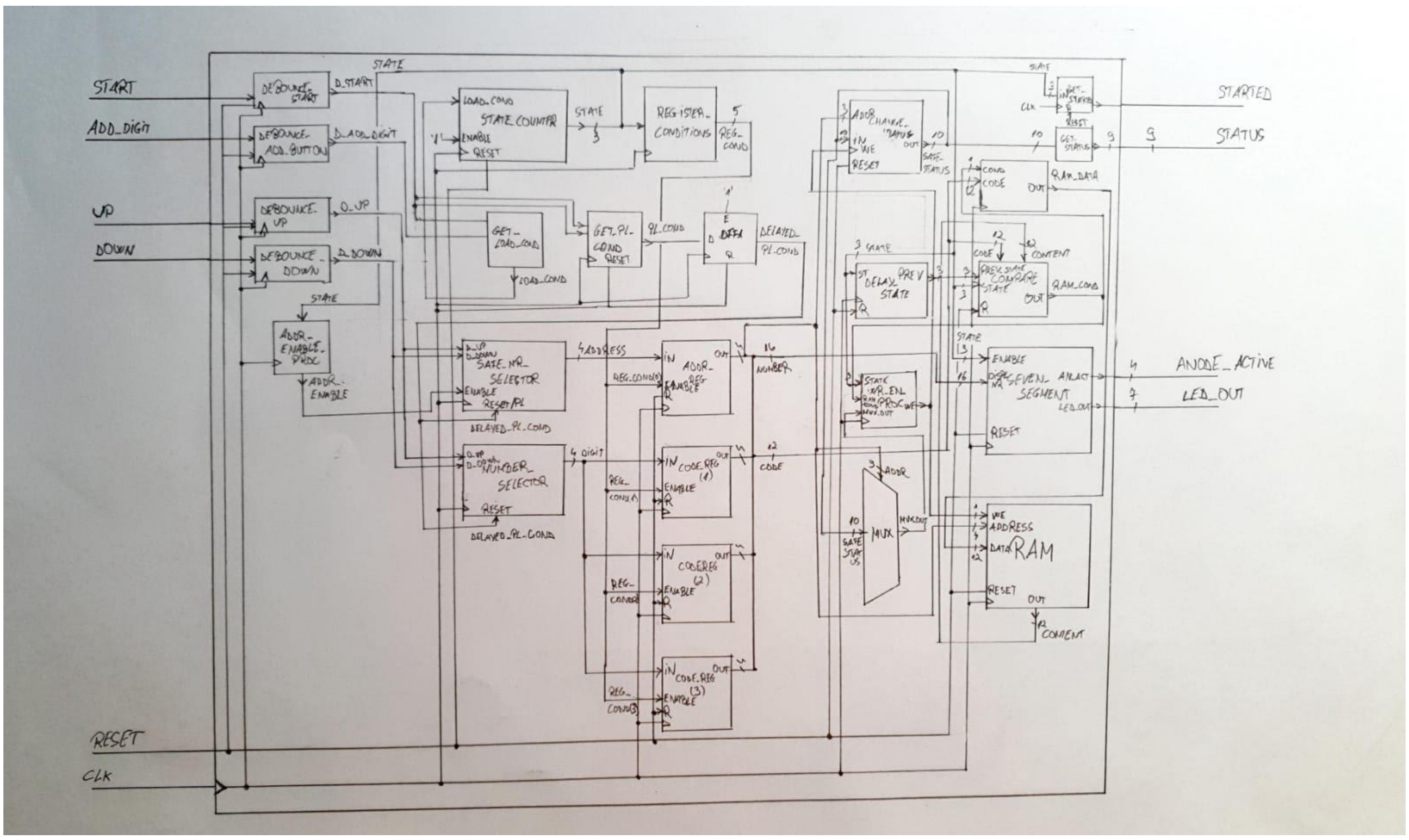
- State is “000” and start = ‘0’
- State is (“001” or “010” or “011” or “100”) and add_digit = ‘0’

This process is done by a the GET_LOAD_COND circuit, while the transition between states is controlled by the state_counter.

Additionally, when state is “001”, the address counter should be enabled. Otherwise, the digit counter should be enabled, therefore, the circuit addr_enable_proc decides the enable for address counter. These circuits help in deciding which register should be enabled, in other words, which of the 4 digits was entered.

After all the digits are entered, the memorize sequence of processes will be executed, which is the most important part of the circuit, because it analyses the input and decides whether it should open the safe, close it or leave it unchanged.

Block Diagram



At first sight, it can be clearly seen that the circuit is a *synchronous automaton*, because each part of it is controlled by a *clock* signal.

Each input button should be debounced because the buttons' signals are not perfect, they oscillate when pressed. That is done via the *debounce* circuit by comparing the input at 2 different times and keeping the output '1' while they are different, so it produces the result '1' only for a short period of time. This will come handy when using as enable at other circuits.

The *safe_nr_selector* is a counter enabled when the user enters the address of the safe, counting in range of 1-9. The *number_selector* is a counter that engages with the code entered by the user, that is not the address, and is a 0-15 reversible counter.

Their *count_up* and *count_down* enable inputs are connected with the up, respectively down debounced inputs. It is practical to use counters for these circuits since the user can navigate between digit only one step at a time and only the count enables have to be selected correctly.

These circuits' outputs are then saved in 4 registers, that represent each digit entered by the user. These registers, named *addr_reg*, *code_reg(1)*, *code_reg(2)*, *code_reg(3)*, have their enables connected to the *register_condition circuit*, that is a decoder for the *state*. The registers are used to remember the numbers entered and we should only one at a time and that depends on the current state.

The registers' outputs are collected by the bus named *number* and the three digits of the code by the bus named *code*.

The *code* is used by the *compare* circuit, that compares the code entered with the one written in the RAM. Its output, *ram_cond*, will be used to determine, whether the code entered is correct or not, in case when the status of the safe is closed ('1'). This, together with the *mux_out* signal will determine the *write_enable* for the RAM, as well as the enable for the change *status circuit*.

The *mux_out* signal is the output of a multiplexer whose inputs are the safes' statuses, and its selections are the address.

The RAM memory stores the codes of the safes, and its input data is determined the following way: if ram_data is '1' then it will be 000H, otherwise, it will be the value of the code. In other words, if entered code corresponds to the content of the RAM at the current address, the content of the ram should be changed to 000H and the safe should open. Otherwise, the content should remain the same.

This depends also on the current state, that's why *write_enable* depends on the *mux_out*, as well.

Further Improvements

A useful improvement could be generalizing the circuit. This would mean that there wouldn't be only 9 cabinet, but n cabinets. In this case, the number of statuses, the address selector and memory had to be adjusted and implement generics in the respective circuits.

A nice addition would be introducing a code of arbitrary length. In this case it is not enough to have 4 registers to store the data and would be quite hard to choose the right number of registers and probably we don't need to modify the length of the code after creating the automaton.

Bibliography

Project GitHub repository