Safe Manager

~Documentation~

Szabó Loránd

Table of Contents

1. *Description* 2
2. *Users' Perspective* 2
3. *Implementation* 3
4. *Block Diagram* 4
5. *Details* 4
6. *Circuit List* 5
7. *Further Improvements* 7

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:

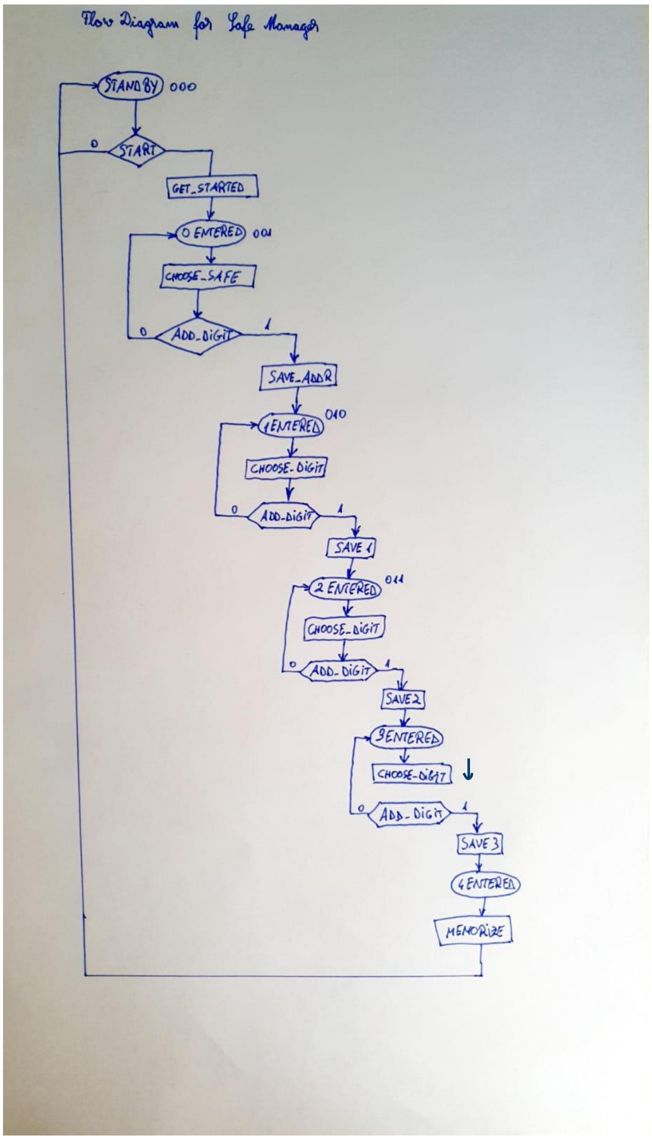
- *Start button:* must be pressed before the user starts to enter the address of the state
- *Add_digit button:* confirms the digit chosen by the user and adds it to the code, also, must be pressed after the sequence of 4 digits is selected (address + code)
- *Up/ Down buttons:* are used to navigate 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:

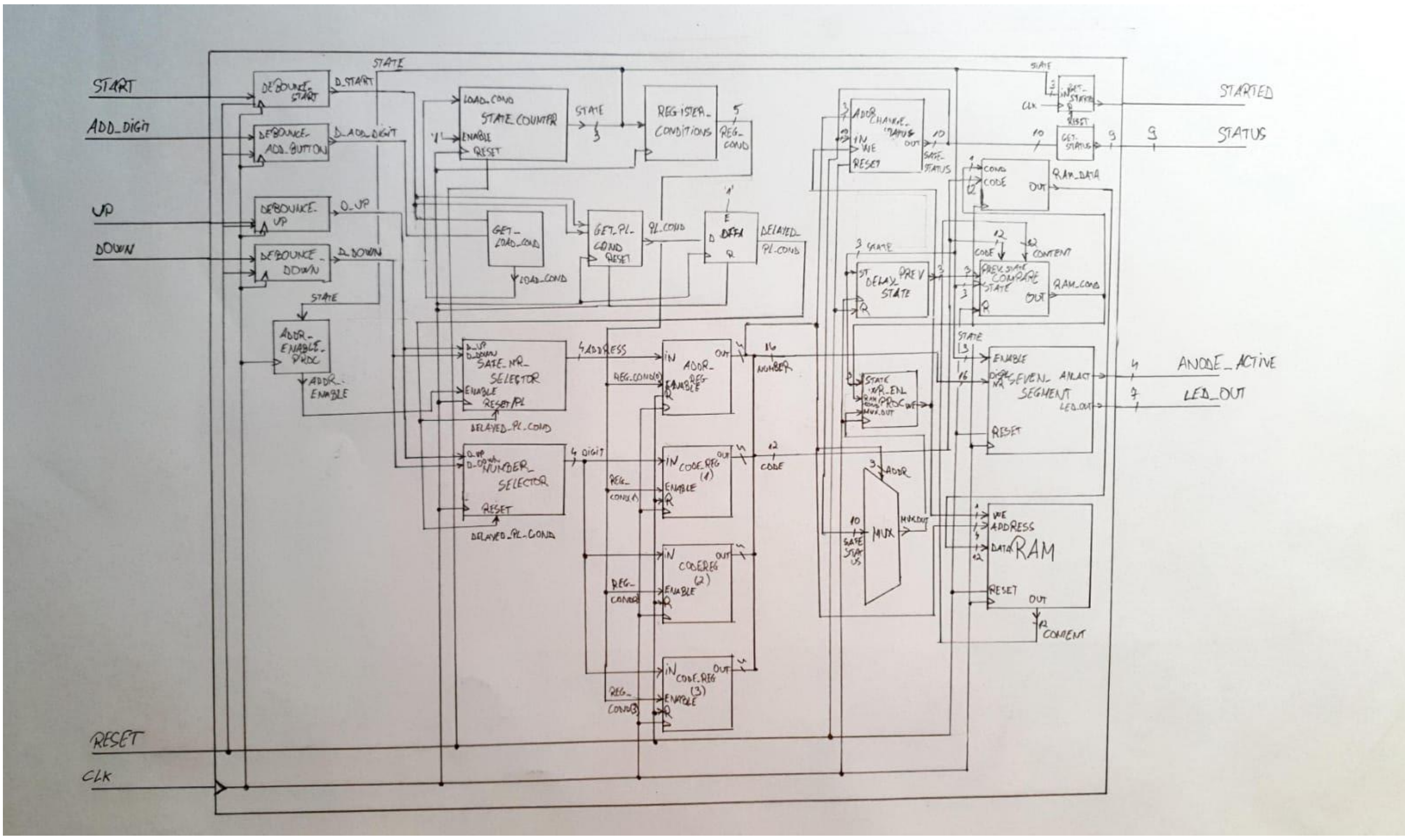
- State is “000” and start = ‘0’
- State is (“001” or “010” or “011” or “100”) and add_digit = ‘0’

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



Details

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.

Circuit List

Circuit	***Inputs***	***Outputs***	***Use***
Debouncers	Button	D_Button	Debounces a button
State_Counter	Load_Cond – synchronous parallel load – from the output of Get_Load_Cond Enable – always one Clk Reset - async	State - 3 bits	Counts the states of the automaton If Load_Cond is '1', then the state remains the same
Safe_Nr_Selector	Up – D_Up Down – D_Down Enable – Addr_Enable from Addr_Enable_Proc Reset - async Clk	Address – 4 bits	Evaluates user input and creates the address from the first digit entered
Number_Selector	Up – D_Up Down – D_Down Clk Reset – Async – Delayed_Pl_Cond from DFF1	Digit – 4 bits	Creates the digit entered by the user using the up and down buttons
Addr_Enable_Proc	State – 3 bits	Addr_Enable	Decides whether to enable the safe_nr_selector,

			corresponding to the current state
Get_Load_Cond	D_Start D_Add_Digit	Load_Cond	Used by the state machine to hold its state
Get_Pl_Cond	D_Start D_Add_Digit Clk Reset	Pl_Cond	Outputs goes into a DFF1 to delay the parallel load of the digit and address selectors.
Register_Conditions	State – 3 bits Clk	Reg_Cond – 5 bits	Basically, a decoder for the state vector
DFF1	D – Pl_Cond from Get_Pl_Cond Clk Reset	Delayed_Pl_Cond	Output used by the safe_nr_selector and digit_selector
Addr_Register	In - Address Enable – Reg_Cond(0) Reset Clk	Out – Number(15 downto 12)	Stores the address entered by the user in the vector Number that is displayed on the SSD
Code_Registers	In - Digit Enable – Reg_Cond() Reset Clk	Out – Code (11 downto 0), that is Number (11 downto 0), as well	Stores the 3 digits entered by the user and creates the Code vector and the rest of the Number vector using them.
Addr_Change_Status	Address In –status WE – write enable – output of wr_en_proc Reset - async	Out – changed status	Changes the status vector
Delay_State	ST – state (3 bits) Clk Reset	Prev – 3 bits	Memorizes the previous state, used by the Compare circuit
Wr_En_Proc	State – 3 bits Ram_Cond – result of Compare Mux_Out – output of the MUX	WE – write enable for change_status and RAM	Decides whether the memory should be overwritten or not
Mux	Safe_Status (10-bit wide) Selection - Address (4 - bit)	Mux_Out – status of the register with index Address	Gets the status of the safe at index Address, entered by the user
Get_Ram_Data	Cond – output of Compare	Ram_Data – number that will enter the data input of the RAM (12 bit)	According to the condition, will put 000H or the code on the Ram_Data bus

	Code – Code – from the 3 Code Registers (12 bit)		
Compare	Prev_State – output of delayed_state State Code – number entered Content – of the RAM Reset	Ram_Cond – goes into wr_en_proc	Compares the entered code and the content of the RAM at the address selected, that is needed to decide the write enable
Seven Segment	Enable – State (3 bits) Displayed_Number – Number (16 bits) Reset	Anode_Active (4 bits) Led_Out (7 bits)	Displays on the SSD the address and the 3 digits.
RAM	WE – from wr_en_proc Address Data – Ram_Data from Get_Ram_Data Reset Clk	Out – Content – goes into Compare	Manages the safes' codes (modifies and reads) and returns the code of the safe at the address entered
Get_Started	State – 3 bits Clk Reset - async	Started - LED	If state is not "000" (initial state), then its output will be '1'
Get_Status	In - Safe_Status (10 bits)	Out – Status, global output (9 bits)	Its output shows whether the safes are open or not

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*]