Signed Fixed-Point Number Calculator

Yusuf Husen Khatri, Peter Barkho, Christopher MacKenzie, Eduardo Garcia
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
ykhatri@oakland.edu, peterbarkho@oakland.edu, cmackenzie2@oakland.edu, eduardogarcia@oakland.edu

Abstract - **The goal of this project was to develop a signed fixed-point number calculator on the Nexys A7 FPGA Board with VHDL. Due to the sheer number of use cases that calculators have across several fields and industries, their importance and implementation should be well understood. Throughout the course of this project, the team members gained much more experience and knowledge of how calculators function, how to interface with the keyboard peripheral, how to output results to a display, and finally how to design a datapath controlled by a finite state machine. In the end, the team was able to successfully design a system that functions as a signed fixed-point calculator.**

I. INTRODUCTION

Calculators are among the most important and highly used tools across many fields, including mathematics, engineering, and science. The main reason being that calculators can perform operations much more quickly than a single human could. This tool heavily reduces the time taken on computing complex calculations by hand and allows for more time for humans to do research and advance in their fields. As such, this team had an interest in learning more about the design of such tools. Therefore, the goal of this project was to utilize skills gained from this course to design a calculator that is able to perform calculations with signed fixed-point numbers. This calculator will be able to perform each of the following arithmetic operations: Addition, Subtraction, Multiplication, and Division. The team will utilize the on-board seven-segment display in order to output the result, and a keyboard in order to read in input from the user. The user will be able to input two fixed-point numbers in hexadecimal format and select an operation to compute the calculation. Each of the input operands can be up to 16 bits, where the integer and fractional portions of the operands will be decided by the user by placement of the decimal point. The calculator will perform the operation and display the output on the 7-Segment display in hexadecimal format with the correct decimal point placement.

II. METHODOLOGY

We have designed the system to work as follows: The user enters the first operand with the hex values from the keyboard (F, E... 2, 1, 0). Within the first operand, the user should enter a decimal point (#.###, ##.##, or ###.#). As the user is inputting values, the values should show up on the 7-segment display. Once the first operand is complete, the user should then press an operation on the keyboard (+, -, x, /). Upon doing this, the user should enter the second operand with the hex values on the keyboard in a similar way as the first operand. Once all inputs are complete, SW0 can be toggled ON and the system will show the result of the calculation. From here, the user must press the CPU reset button to clear the display and start from the beginning with the next calculation.

The components utilized in this project include a PS2 Keyboard Decoder, an Arithmetic Operations Circuit, several Registers, a 7-Segment Display Serializer, several multiplexers, and finally a Finite State Machine. Each of these components will be explained in further detail in subsequent sections

A. Keyboard Decoder

The keyboard decoder is a circuit that reads an eight bit signal from the standard component "my_ps2keyboard" [1] and outputs four signals. my_ps2keyboard is a standard component developed and provided by Professor Daniel Llamocca. This component waits for a ten bit input signal from a PS/2 keyboard and then outputs an eight bit signal that corresponds to the hex value of the corresponding key that was pressed. The timing and validity of the keyboard input is handled within my_ps2keyboard by waiting for a valid length input, and validating a stop and parity bit within the input. When these bits are correct, the component outputs the desired eight bit sequence, which is tied directly to our designed keyboard decoder. The outputs from the decoder are: done, output_number, output_operation and output_dp. The 'done' output signal is tied directly to the 'done' output signal from the component my_ps2keyboard. Done is pushed high again when a valid stop and parity bit are captured, which means the keyboard input data is correct and ready to be decoded. The second output signal output_number is the hexadecimal value of the decoded input data from the keyboard. The eight bit input "DOUT" is read from the component my_ps2keyboard. The keyboard decoder then uses these eight bits as the data from the keyboard without having to manage timing and parity. These eight bits are then compared using a case statement.

The case statement outputs the hexadecimal value of the corresponding key represented by the keyboard input data. Please view the Appendix for PS2 keyboard values. This section of the decoder selects for the keystrokes: 0-9 and A-F. All other keystrokes are decoded as "0000". This hex value then goes on to be stored and used for calculations within the arithmetic operations circuit. The third output signal, output_operation functions very similarly to the output_number signal. The keyboard input data is read from the my_ps2keyboard component, then is converted to an arbitrary two bit value. This arbitrary value is used later on in the project and is selected for these specific values, so decoding into the hex values of the operational input is not necessary.. In this case the decoder decodes the values of the operands (*, /, +, -) from the keyboard input data. The decoded value will then go into the arithmetic operations circuit to select which operation will take place. Lastly, the output signal output_dp. This signal again functions the same as the output_number signal. The difference is this part of the decoder is looking for the corresponding input data of the decimal point from the keyboard. The decimal point output signal is a single bit signal and goes on later to determine where the decimal point is present in the operations. This signal is tied directly to an input into the FSM.

B. Arithmetic Operations Circuit

The Arithmetic Operations Circuit is a circuit which is capable of performing the addition, subtraction, multiplication, and division of signed fixed point numbers. It is essentially composed of two addsubs (one for addition and one for subtraction), a signed multiplier (based on an array multiplier), and a signed divider (based on an iterative divider) [2]. Each of these individual components are available from the notes and VHDL tutorial. The Arithmetic Operations Circuit takes in both operands as inputs, where each operand is split up into four 4-bit inputs. This means that the circuit is able to operate with 16-bit operands. Additionally, this circuit takes in a start signal, the operation you would like to do (2-bit input), and the decimal positions in each of the operands (input as an integer). As output, the circuit will give the result of the calculation as a 32-bit number, as well as an integer representing the decimal point position in that number. In terms of the actual calculations, this circuit also performs alignment on the input operands. Due to having different fixed-point representations as input, we perform alignment for the addition and subtraction such that the output is always of the form [32, 16] (16 integer bits and 16 fractional bits). For multiplication, the alignment does not matter very much. However, the input decimal positions help to determine the output decimal position. Finally, for the division, we perform alignment and utilize 4 precision bits. These precision bits allow for a consistent output in terms of decimal point position.

Thus far, this description has been a high level overview of the Arithmetic Operations Circuit. However, it is also important to understand the inner workings of it. The main characteristic of this circuit which is crucial to signed fixed-point numbers is the alignment of the operands. If one were to perform a calculation without proper alignment, then the result will almost never be correct. As stated previously, the alignment of the operands largely depends on the decimal point positioning of the numbers. One possible method of alignment is to look at the decimal point positions relative to each operand and try to match the other. While this would work, the implementation of a solution like this could get messier. The solution that the team members utilized in this design was to effectively, as stated previously, make both operands into 32-bit numbers through sign-extension and zero-padding. This would be done relative to the operands own decimal point positioning. For example, suppose the user has inputted a 16-bit operand of the following fixed-point form: [16, 12]. This implies that the number looks like this: "#.###". In this case, in order to make this number into [32, 16], the number is sign extended by 12-bits using the MSB and the number is zero padded by 4 bits after the LSB. When the operand is of the form [16, 8] ("##.##"), the number is sign extended by 8 bits using the MSB and the number is zero padded by 8 bits after the LSB. When the operand is of the form [16, 4] ("###.#"), the number is sign extended by 4 bits using the MSB and the number is zero padded by 12 bits after the LSB. When the number does not have any fractional portion, then the number is zero padded by 16 bits. This alignment is done to both operands and is crucial for addition, subtraction, and division. It essentially allows for a consistent and deterministic output by making both operands of the same form.

Another important aspect of the Arithmetic Operations Circuit is the determination of the output decimal point position. In the case of addition and subtraction, due to the fact that the alignment was done beforehand, the decimal point position will always be in the 4th position (“####.####). This is another benefit to performing alignment this way - it allows for consistent decimal point positioning for the result. For multiplication, the output decimal point position would be the sum of the two input decimal point positions. For example, if the input decimal point positions were 2 and 3 for operand 1 and 2, respectively, then the output decimal point position would be 5 (“###.#####”). For division, the team decided to utilize 4 precision bits. Therefore, the output result will always have a decimal point position of one (“#######.#”). This is consistent and a byproduct of how fixed-point division works. If we were to have an increased number of precision bits (8, 12, etc.), then the decimal point position would simply move, but it would always be consistent.

C. *Registers*

The register in our design is a basic N-bit register that stores input data from the user and also has the ability to copy/output data to other components of the system. The register consists of an “enable” function that syncs with the clock in the finite-state-machine in order to start the process of storing/outputting data. It also consists of a synchronous clear function which can clear all the data stored in the register. There are a total of 23 registers in the design of the system. The registers are enabled independently by their own enable signals directly controlled by the FSM. The FSM first enables the registers responsible for storing the input data from the user chronologically as the user presses correct keys on the keyboard. The second set of registers responsible for storing the calculated values are all enabled simultaneously after the arithmetic circuit signals that it has completed the calculations i.e., signal *calculate* is pushed high. The registers responsible for storing the decimal point placement and the operation are enabled similarly to the input data registers. When the user inputs a decimal point or an operation command, the FSM enables the corresponding registers. One of the registers is a 2-bit register that is responsible for storing the operation type(s) that the user selects for the calculation. 8 of the registers are 4-bit registers that are used to collect the user’s input for the 1st and 2nd operand. 3 other 1-bit registers are used to store the decimal point placement given the user’s input. These 12 registers output data to the finite-state machine in our design where the calculation is performed. Furthermore, the 11 other registers are used to store the output result of the calculation. 8 4-bit registers are responsible for the output result after the calculation is performed, and 3 1-bit registers are used to determine the decimal point placement of the output result.

D. *Seven-Segment Serializer*

This particular 7-Segment Serializer is being fed by eight, 2-to-1 Multiplexers. The subcomponents contained within the Serializer are the following: HEX-to-7 Segments Decoder, 3-to-8 Decoder, Counter, and its own unique Finite State Machine. The ‘serializer’ component [3] utilized is almost similar to its version of Lab 3’s Design of an Accelerometer Data Retriever. However, the serializer.vhd and hex2sevenseg.vhd [4] files were modified so that they take 5 bits as input and the DAT_7SEG output is of 8 bit length. The reasoning behind this additional bit is to handle the decimal point. Due to this system being a calculator for signed fixed-point numbers, displaying the decimal point is important. By displaying the 8 digits from inputs A to H, we will be capable of doing so by serializing the HEX digits through the decoder. The Counter integrated enables the digit to be illuminated constantly by a generic component. With a behavior on the clock tick of 1 millisecond, will allow the state transitions to occur at that particular time set, which brings us to the next component, the 3-to-8 Decoder. The input signals and the enable signals to the eight, 7-segment displays are active low from the decoder, which are derived from the FSM and the multiplexer. The HEX to 7 segments decoder enables the LEDs to illuminate the digits from ranges, one through nine and A through F, twice, followed by a decimal point for one set in case it’s needed. Last, but not least a Finite State Machine of 8 states is implemented into the serializer feeding into the 3-to-8 Decoder. Which as, for every state there is a particular logic designated to execute whenever it’s required. Please view the Appendix for the modified serializer diagram.

E. *Multiplexers*

The main purpose of the several multiplexers that are used in this design was to select between showing the user input on the 7-Segment Display while they are inputting and the arithmetic output on the 7-Segment Display when the calculation is complete. There are a total of eight two-to-one bus multiplexers. The inputs of these multiplexers are tied to the corresponding registers for each digit on the seven segment display. These signals are declared as "DATAIN-DATFIN" for the user inputs and "DATAOUT-DATFOUT" for the calculated values. These connections can be further visualized in the full block diagram (Figure 1). The outputs for the multiplexers are tied directly to the A-F input bus signals in the 7-segment serializer. These signals can be further explored in the Seven-Segment Serializer section. The multiplexers utilize the 'sel' signal which is controlled by the Finite State Machine. The 'sel' signal is tied to switch 0 on the nexys board. This allows the user to quickly switch between showing the current input and the calculated output. When 'sel' is zero, then the Seven-Segment Display will show the input values whereas when 'sel' is one, then the Seven-Segment Display will show the output values.

F. Finite State Machine

The Finite State Machine (FSM) houses much of the logic that occurs when the user is inputting their operands, operations, and decimal points. There are a total of 19 states in this FSM. The first eight states are designated for the user to input their first operand, including the decimal point position. The following two states are designated for the user to input their operation of choice - addition, subtraction, multiplication, or division. The next eight states are designated for the user to input their second operand, including its decimal point position. The states up to this point are fairly similar to each other. They essentially ensure that the user is properly inputting things from the keyboard at their designated times. The final state simply displays the output on the 7-segment display when the arithmetic operations circuit is done operating and SW1 on the board is toggled on. From this point, the user can press the reset button in order to clear the display, return to the first state, and start another operation. Please view the Appendix for the diagram of the FSM.

Going into detail on the individual states, the Finite State Machine in this design begins in S1. In S1, the user inputs the first hexadecimal portion of operand 1. The logic of the FSM prevents the state to continue unless the user has inputted a proper value. In the case of an erroneous input (such as a non-hexadecimal character like 'S', for example), then the FSM will assume that the input is hexadecimal 0. In the next state, S1a, the logic is simply to make sure that a single key-press has occurred for the first portion of operand 1. The following states, S2-S4 (and their corresponding intermediary states) are very similar to S1. The only difference is that the user is also able to input a decimal point within these certain states. When a user inputs a decimal point position, then the logic checks to make sure that a decimal point makes sense here (the user has not inputted another decimal point previously). If that logic checks out, then the enable for the decimal point register is driven high in order to capture that position. The decimal point position input is not enough to move to the next state, so the FSM stays within its state until the user has inputted a hexadecimal value for the operand. After S4 and S4a, the Finite State Machine moves on to Sop, a state designated for the user to input their operation of choice. In this state, the user can not enter a hexadecimal number or a decimal point, but rather only a correct operation. If the user inputs an erroneous value here, then the system will simply assume that the user has selected addition. After Sopa (the intermediary state for Sop), then S5 is entered. States S5-S8 work similarly to S1-S4 (along with their intermediary states). After S8a, the FSM will enter S9, which is the final state. The FSM will start the arithmetic operations circuit here and wait for it to finish. Once finished, the user is able to control the 'calculate' signal with SW0 to show the final output on the Seven-Segment Display. Finally, as stated previously, the user has now finished their calculation and can press the reset button in order to clear the display and start again from S1.

G. Signed Fixed Point Calculator Top File

Each of the smaller components were eventually interconnected in a top file - my_fx_calculator.vhd. In order to view the block diagram of the entire top file, please view the Appendix. The main inputs of this top file were the 'clock' and 'resetn' signals, 'ps2c' and 'ps2d' for the PS2 keyboard interfacing, and a 'calculate' signal which is directly controlled by SW0 on the FPGA board (used by the user to show the output result or not). The main outputs of this top file are the 'AN' and 'DAT_7SEG' which are directly related to the outputs of the seven-segment serializer. Using these two outputs, the seven-segment display was able to show the inputs that the user was entering with the keyboard and the final result of their calculation. In addition, this top file contains the VHDL code for the Finite State Machine and its logic. The team decided to develop the Finite State Machine in the top file itself due to stylistic reasons. It could have been a separate file/component, but it was included in the top file itself since it made no difference on the resulting operation of the system.

III. EXPERIMENTAL SETUP

The experimental setup for this project involved creating testbenches for the main components of the system, such as the arithmetic operations circuit. By testing these components with simulation, the team members were able to verify that the system should technically work. However, the system did not initially work on the board despite the results of the simulation of the components. After much testing, the team was able to find the hidden issues with the design and resolve them in order to make the system work on the FPGA. After verifying that the system was working on the board, the team was able to test various computations with the calculator to ensure that the proper results were being achieved for addition, subtraction, multiplication, division, different decimal point positions, positive and negative numbers, and erroneous inputs from the keyboard.

IV. RESULTS

As a result of testing the design of the system, the team was able to verify that the system behaves as expected. In order to show that the system is working properly, the team has put together various test cases. These test cases are shown below for each operation:

Addition: 37.AB + 1.FC8 = 0039.A780
Subtraction: F.540 - 682.5 = F97D.0400
Multiplication: 3D.21 x 6.CF7 = 1A0.4E6D7
Division: FFE.6 ÷ 000.7 = FFFFFFC.5

In addition, these test cases were recorded and are available in the following video to show that these cases truly work: https://youtu.be/milBWN9lGGo

CONCLUSIONS

In conclusion, the team was ultimately able to meet the requirements set out initially for the project. That is to say a working signed fixed-point calculator was successfully designed and implemented on the Nexys A7 FPGA Board. Many important lessons were learned throughout this development process, such as how to interface with a PS2 keyboard to read and decode user input, how to output data onto a Seven-Segment Display with a decimal point, and how to design a Finite State Machine to incorporate logic within systems. In addition, the team gained a much better understanding of how calculators are created and designed.

While the system behaves as expected, that is not to say it is a perfect system. Given additional time, several improvements could be made to the system to make it an overall better calculator. For example, this calculator is only able to operate with 16-bit operands. This is a limitation due to the fact that there are only 8 Seven-Segment Displays available on the board. If there were more 7-Segment Displays available on the board, then the user would be able to input a much larger number and perform large calculations. An improvement that could be made to solve this problem would be to instead utilize an LCD as a display. This way, the user is not limited by the number of Seven-Segment displays available on the board. While this would make the design and logic of the system more complex, it is certainly a route that could be made in the future. Another possible improvement would be to allow the user to select between signed and unsigned fixed-point numbers. This could easily be done in the future and allow for more versatility of the system, as the components for signed and unsigned arithmetic are very similar. Lastly, a logistical improvement could be made to the keyboard decoder component. Currently the decoder selects for the expected keypresses 0-F, operands, and a decimal point, while pushing all other inputs to “0000”. This may cause an issue when a user mistakenly presses another key. An improvement could be made to allow the decoder to see these keystrokes, but not assign any value to the decoded output. This would allow the user to press the correct key even after pressing an invalid key, rather than having to reset the system to correct their mistake.

Overall, the team was able to successfully implement a signed fixed-point number calculator on the Nexys A7 FPGA Board.

REFERENCES

[1] Llamocca, Daniel. *VHDL Coding for FPGAs*, Oakland University, “my_ps2keyboard.vhd”

[2] Llamocca, Daniel. *Unit 2 Notes*, Oakland University, Multiplier and Divider

[3] Llamocca, Daniel. *Laboratory 3*, Oakland University, “serializer.vhd”

[4] Llamocca, Daniel. *Laboratory 3*, Oakland University, “hex2sevenseg.vhd”

APPENDIX

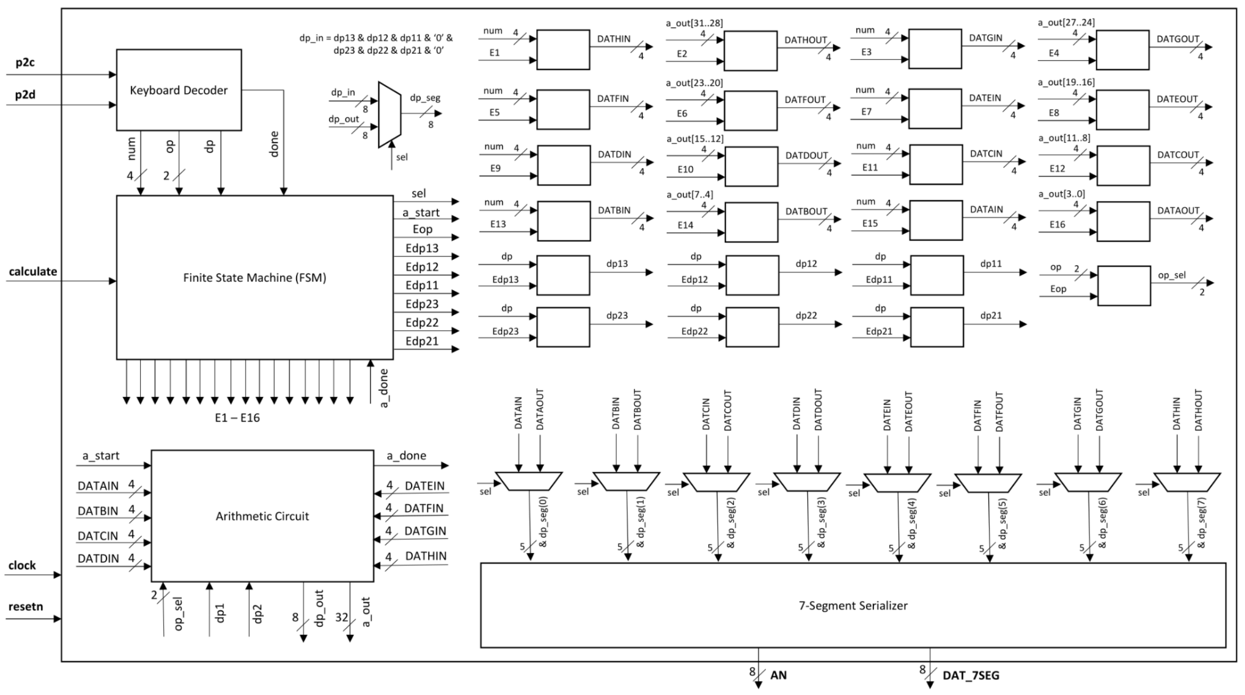
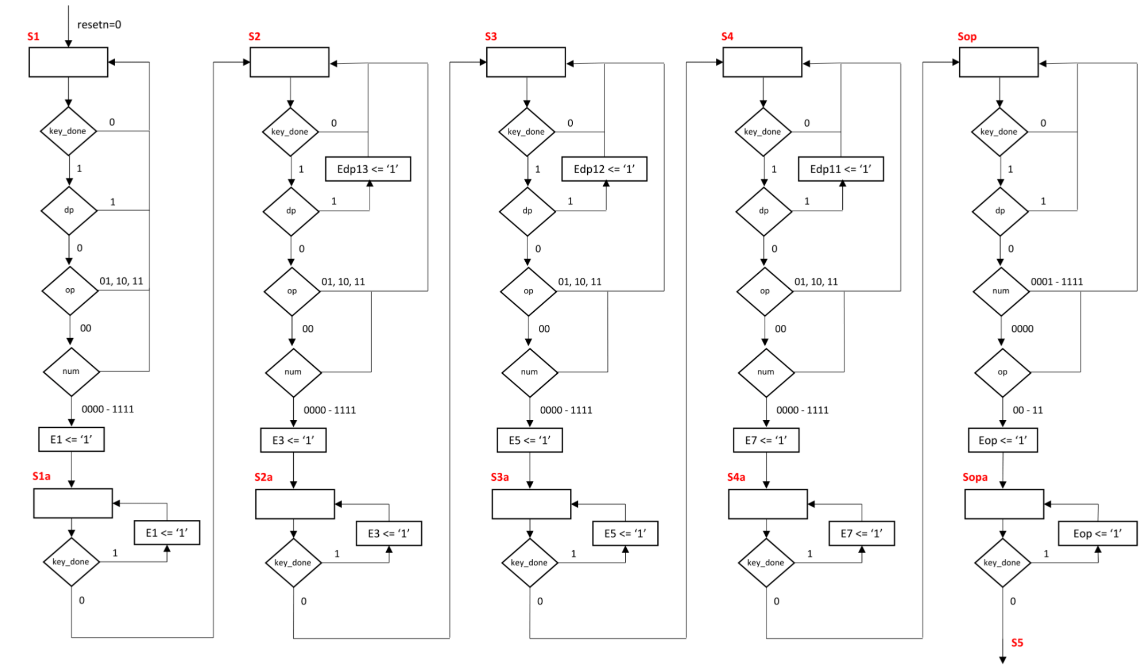


Figure 1. Block Diagram



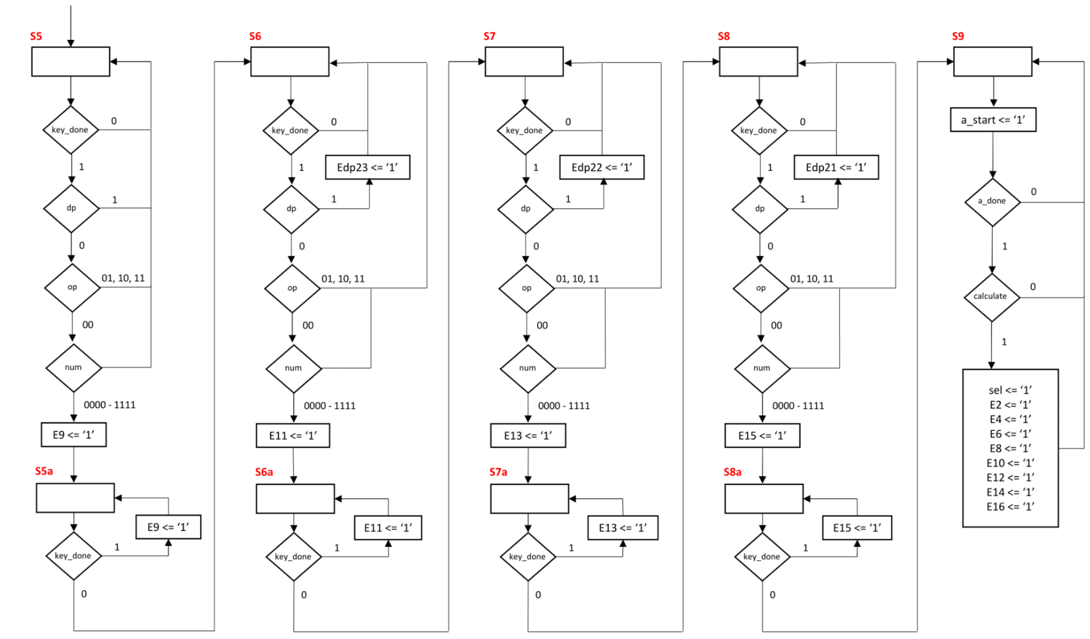


Figure 2. Finite State Machine

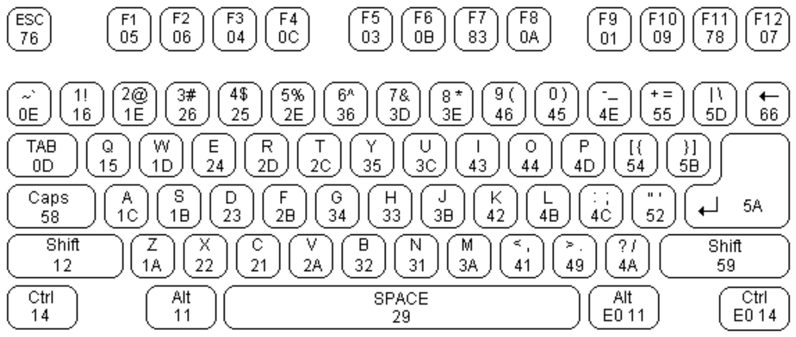


Figure 3. PS2 Keyboard Values

- **8-display Serializer**: Eight 7-segment displays.

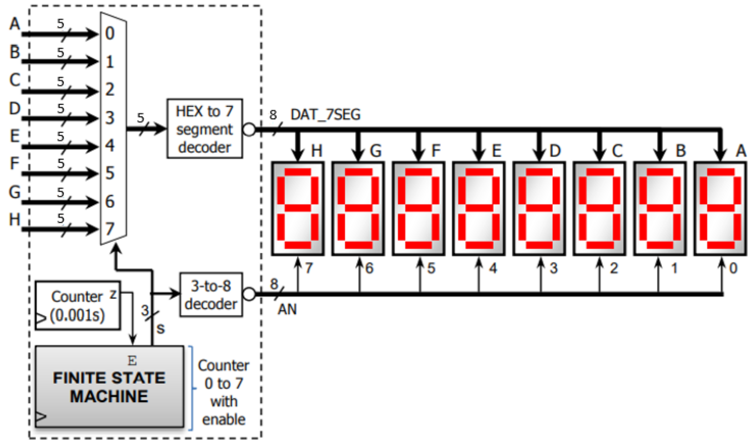


Figure 4. Modified 7-Segment Serializer with Decimal Point Feature