# Fixed-Point Calculator

Danijel Spasic, Rami Sulaiman
Professor: Daniel Llamocca
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: dspasic@oakland.edu, ramisulaiman@oakland.edu

**Calculators are extremely useful components in most any field. They have a wide application and are utilized in many areas in today's world. Thus, the goal of this project is to create a fully functioning calculator for signed fixed/non-fixed-point numbers. Through this project, a much better understanding of PS/2 interfacing was established, along with a greater understanding of how to build datapath circuits and design finite-state-machines.**

## I. Introduction

Calculators are one of the most efficient tools ever created. Their ability to perform operations of large numbers with lightning speeds makes most all mathematical calculations seem like a breeze. However, it is very interesting to consider how a calculator with all of its components works together to output desired values. Therefore, the aim of this project is to create a fully functioning calculator for signed fixed-point numbers. The final result of this project will allow the end-user to perform basic arithmetic operations on signed fixed and non-fixed-point numbers. The user will be able to input hexadecimal numbers and perform basic arithmetic operations (addition, subtraction, multiplication, division) as on any given calculator through a keyboard. The input numbers will be limited to a 16-bit width. The inputs and outputs will be visible on a 7-segment display.

## II. Methodology

The project was split up into various parts, in order to simplify the process. The inputs from the user are read as signed two's complement numbers, and the outputs are to be interpreted as signed two's complement numbers. The input operands are 16-bits wide, and the output will thus be up to 32-bits wide. The inputs and output are displayed on the on-board 7-segment display. The input peripheral is a regular USB keyboard. The Fixed-point Calculator Datapath can be seen within the References section (*Figure-5*). This datapath circuit will be discussed in detail in the sections below, describing every component within this circuit.

### *A. Keyboard Interfacing*

The file for interpreting the user keyboard inputs was provided by the professor (*my_ps2keyboard.vhd*). This file outputs a "done" signal once a key is pressed, along with a specific 8-bit code for each keypress. This code is then converted into a value utilizing a decoder (*keydecoder.vhd*). This decoder only considers specific keys on a generic keyboard (see References section, *Figure-3*, for key specification).

The decoder will output a 7-bit code: the three MSBs (most-significant bits) refer to either a dot press or an operation selection, and the four LSBs refer to the value of the number pressed (0, 1..E, F). The four LSBs outputted by the decoder will be tied to the eight registers.

### *B. Registers*

The registers play a major role within this project. They have several inputs and outputs which function to output the required values to the serializer component. The provided (*my_rege.vhd*) register file was manipulated in order to give the registers additional functionality required for this application.

Regarding their inputs, all of the registers will be inputted six hexadecimal values: the input key value pressed (*keydecoder.vhd* output: four LSBs) and the results (4 total) of the arithmetic operation component (these results will be further discussed in the Arithmetic Operation Circuits section). In addition, the registers will have the basic clock and reset input signals, as they are made to only operate on rising edges of clock cycles and are created with the ability to be reset if desired. Also, the registers all have an enable input signal, along with a synchronous clear signal (for this case, the synchronous clear signal was unneeded and thus driven with a constant low). With only the current signals discussed, while these registers would be able to function and display results, they would not be able to display the required information. Therefore, three additional signals were added: result, operation, and dot. The result signal will determine whether or not the results of the arithmetic operations are to be outputted. The result signal works in conjunction with the operation signal, which is three bits wide. The operation signal will tell the code which of the arithmetic operation results to output. Lastly, the dot signal will place a '1' or a '0' on the LSB (least-significant bit) of the output value "Q". For example, if the dot signal is high, then the output of the register will have a '1' added at the LSB or concatenated as the LSB for certain cases. All three of these signals will be driven by the logic finite-state machine.

In terms of the outputs of the registers, they will only be outputting two five-bit wide numbers. These numbers are both the user inputted hexadecimal numbers, and the arithmetic operation results. Both will be tied to the serializer component.

Two very similar registers files were created for this project: *my_rege.vhd* and *my_rege_firstvalue.vhd*. The file *my_rege.vhd* was utilized for registers reg1 thru reg3 and reg5 thru reg7. Due to the logic of the finite-state machine, another file had to be created for registers reg0 and reg4. This file was called *my_rege_firstvalue.vhd*. This file will allow for a hexadecimal point bit to be concatenated on the first hexadecimal number of both operands. The difference between the two register files created (*my_rege.vhd* & *my_rege_firstvalue.vhd*) is the fact that the *my_rege_firstvalue.vhd* file will allow a hexadecimal point to be written to the first register of both operands (reg0 & reg4) for the specific case of the hexadecimal point being the very first key press.

### *C. Serializer*

The file for writing numbers onto the 7-segment display was provided by the professor (*serializer.vhd*). However, this file only displays values on four of the eight displays, starting from the right-most display; the input values populate from right to left on each display. In addition, the *hex2sevenseg.vhd* component did not take into consideration the decimal point. Therefore, modifications were required. First, the *serializer.vhd* file was modified in order to enable all of the displays. This allowed for four more inputs to be added to the multiplexor. In addition, the orientation of the displays was rearranged so that the inputs A down to H would be arranged from left to right. Also, the inputs were made to be five bits wide, with the LSB being the driver for the dot (dt) segment of each display. Thus, the *hex2sevenseg.vhd* file was edited to account for the dot, as well. See References section for the serializer datapath circuits (*Figure-2A* & *Figure-2B*).

The last edit made to the *serializer.vhd* file was adding another set of eight inputs, along with a signal which was labeled "result." This second set of inputs will be connected to the result outputs of the registers, as opposed to the operand value outputs which are connected to the first set of eight inputs (A, B…H). The way this circuit now works is that once the signal "result" (which is an output of the finite-state machine) is high, the serializer will display the result outputs of the registers. This will allow for the user to be able to see the outcome of the selected operation. Thus, after making the required edits to the *serializer.vhd* and the *hex2sevenseg.vhd* files, this new circuit could now be utilized as required.

### *D. Arithmetic Operation Circuits*

The addition and subtraction files utilized were created by the professor (*my_addsub.vhd* and *fulladd.vhd*). No changes were made to these files. The multiplication file utilized was also provided by the professor (*my_mult.vhd*). No alterations were made to this file either. The unsigned division circuit was created utilizing the datapath circuit seen in *Figure-1A*, and only the quotient was considered. The finite-state machine seen in *Figure-1B* (see References) was also created from scratch. After creating the unsigned division circuit, the signed division circuit was created (see *Figure-1C* in References). All of the arithmetic operation circuits were combined in a top file.

The arithmetic operation component (*arith_circuits.vhd*) is a file that combines the arithmetic operation circuits. This circuit has several inputs and outputs. The more crucial inputs are the outputs of the registers, in particular the user inputted values that get passed by the registers. These inputs are 5-bits wide. When inputted into this component, the first step was to save the inputs in two variables (labeled op1 & op2) without their LSBs (which refer to whether or not a dot is appended with the given value). These two variables are essentially the two inputted operands. The LSBs of the inputs into the circuit (which were originally not considered for the operand variables) will be then utilized to determine where a hexadecimal point is to be placed after each of the operations have been completed. Each of the operations has specific logic which will handle the inputted operands, and then determine where a hexadecimal point should be placed on the result.

The first section of this component deals with the addition and subtraction input operand logic. In this section, operand one was split up into two 16-bit numbers (labeled op1_x & op1_y). The variable op1_x refers to the integer portion of the first operand, while the variable op1_y refers to the fractional portion of the operand. The variable, op1_x, was sign-extended where required, while op1_y was concatenated with lagging zeroes where needed, since the goal was to make both of the variables 16-bits wide. The number of zeroes that was concatenated for the op1_y variable, along with the number of signs by which op1_x was extended by, was determined by the location of the hexadecimal point. This same process was followed for the second input operand. An example of this code can be seen below (note: Q1(0) refers to the LSB of the second input value of the first operand).

```
elsif Q5(0) = '1' then --##.## (2 sign-extension ' 4 bits)
    op2_x <= op2(15) & op2(15) & op2(15) & op2(15) & op2(15) & op2(15) & op2(15) & op2(15) & op2(15 downto 8);
    op2_y <= op2(7 downto 0) & x"00";
```

Now that both of the variables have the same number of integer bits and fractional bits, the two inputted operands are finally aligned. The two variables op1_x and op1_y were concatenated and input into the adder and subtractor components, connecting them to variable "x" of the component. Similarly, the other two variables (op2_x & op2_y) were also concatenated and input into the adder and subtractor components, connecting them to variable "y" of the component. The outputs of the addition and subtraction components were connected to another variable which was then concatenated with zeroes at every single hexadecimal number position, except for the position of where the hexadecimal dot should show up. At this position, a one was concatenated with the number, in order to signify that a hexadecimal point should be displayed. Since it is known that the fractional and integer portion of each number has 16 bits, the hexadecimal point on the output should show up where the integer portion of the output ends, and where the fractional portion begins (i.e. 16-bit positions from the LSB). This concatenation can be seen below.

```
Result_add <= added(31 downto 28) & '0' & added(27 downto 24) & '0' & added(23 downto 20) & '0'
--segs:              A                        B                        C

       & added(19 downto 16) & '1' & added(15 downto 12) & '0' & added(11 downto 8)
                 D          point          E                        F

        & '0' & added(7 downto 4) & '0' & added(3 downto 0) & '0';
                        G                        H
```

Thus, the final result of the addition and subtraction circuits were the variables "Result_add" and "Result_sub," respectively. These variables end up being 40-bits wide, since they are made up of individual eight 5-bit numbers. Again, the LSB of each individual number will determine whether a hexadecimal point will be displayed with the given number. These outputs are then tied to the eight registers: the MSBs were tied to the first register (reg0) and the LSBs were tied to the last register (reg7).

The next section of the arithmetic circuit component deals with the multiplication variable logic. In this section, the operand variables (op1 & op2) were first passed through the multiplication circuit (*my_mult.vhd*) and a result was obtained. However, this result (32-bit number) needed to be concatenated with a combination of zeroes and potentially a one, in order for the output of this circuit to properly display values and a hexadecimal point (if it exists). The logic for displaying a hexadecimal point on a certain position in the in the "Result_mult" variable was solely based on the position of the hexadecimal point in the user inputted operands. This logic was split up into categories from zero to six, signifying how many hexadecimal places from the LSB position a hexadecimal point should show up in. For example, if the hexadecimal point in the first operand shows up after the first hexadecimal value is inputted ("#.###") and the hexadecimal point in the second operand shows up after its third hexadecimal value is input ("###.#"), then the hexadecimal point has to show up four hexadecimal positions from the LSB of the multiplied output. An example of logic created for these cases can be seen below.

```
--n=6-----------------------------------------------------------------------
if (Q0(0) = '1' AND Q4(0) = '1') then --result =  A  B. C  D  E  F  G  H
       dot_mult <= multiplied(31 downto 28) & '0' & multiplied(27 downto 24)
    --segs                A                              B
& '1' & multiplied(23 downto 20) & '0' & multiplied(19 downto 16) &
 point             C                              D

'0' & multiplied(15 downto 12) & '0' & multiplied(11 downto 8) & '0'
                  E                              F
 & multiplied(7 downto 4) & '0' & multiplied(3 downto 0) & '0';
              G                              H
```

The last section of *arith_circuits.vhd* file deals with the logic for the division of the two user input operands. The first step of this process was to take the absolute value of the inputted operands, and then align the two operands based on where their hexadecimal points were located. This would determine how many zeroes need to be appended to the LSB side of the either operand, along with how many zeroes need to be appended to the MSB for either operand. After the operands were aligned and saved in their respective variables, the first operand variable will be zero-extended with four fractional bits. This value, along with operand two's variable, will be tied to the division circuit in order for the given operands to be utilized for the user selected division operation. The output of the division circuit will be tied to the Result_div variable, in order for the top-file to output the result of the operation. The output of the division file will be concatenated with zeroes and a one at exactly four bit positions to the left of the LSB of the division circuit output. The reason behind this placement is the fact that the operation was performed with four fractional bits. Thus, the dot must show up on the displays before the four LSBs of the division output.

Overall, the *arith_circuits.vhd* file will output all of the required calculations. In addition, the results of all of the circuits will output an error messages ("EEEEEEEE") if the user inputs invalid numbers (i.e. numbers that have more than one hexadecimal point).

### *E. Logic Finite-State Machine*

The finite-state machine depicted in the Fixed-point Calculator Datapath (see References, *Figure-5*) is the brain of the project. This logic circuit drives the serializer and registers which both captures the values input by the user and allows for the registers to output the user designated operation results on the displays. This program operates based on the following inputs: a keypress signal from the keyboard component, a display calculation signal from the board (SW0), and the operation/point code from the keypress decoder component. With these inputs, the logic of this finite-state machine will output the enables to all of the individual registers, dot signals to signify whether the register should display a dot with its output, a result signal signifying that the user wants to see the results of the operation (this will be tied to the serializer as well), and the selected user-defined operation which will tell the registers which of the operation results to display. Now that the inputs and outputs to this circuit were highlighted, it should now be noted how the logic of this component functions.

This finite-state machine has 17 states total and operates only on rising edges of clock cycles. The states S1-S8 each have their own sub-state, denoted by an "a" (S1a, S2a…S7a, S8a). The purpose of these sub-states is to ensure that a user keypress was completed before jumping into the next state. For example, while in the state, S2, the program will only move into S2a if the keypress was a hexadecimal number. While in S2a, the program waits to ensure that the keypress signal is low, signifying that the user has inputted only one number and has removed his finger off the key, before jumping into the next state, S3, and allow for the next hex digit to be inputted. All of the states follow this general logic. However, there is a slight difference in how the first digit input states (S1 and S5) handle the input from the user, as opposed to the other states. This is due to the fact that these states must display a point if that is the user's very first input. For example, in S3, if a user presses the dot key, then the program will still stay in S3 and wait for the user to input a hex digit. However, the program will also send a dot signal (in this case "dot1") to the previous register, in order to display a dot along with the already inputted hex digit. This is how states S2-S4 and S6-S8 function. On the other hand, while in S1 or S5, if the dot key is pressed, the program will not remain in these states and wait for a digit, but rather tell their respective registers to display only the point, and then move on to the next states, for the user to input the next digits.

After the last digit of the second operand is input, the program will jump to the last state (S9). Finally, while in S9, the program will stay in this state indefinitely until the user presses the on-board "resetn" button, restarting the logic back to the first state. In regard to what the program does in this

state, it will either display the results of whatever operation the user selected or display the input operands. In this state, the finite-state machine will output a signal called "result," which will be connected to the registers and the serializer, to tell both of the components to output the result of the operation. It will also output a signal called "operation_selection," which will tell the registers which operation result the user would like to be displayed on the 7-segment displays.

Overall, this component will allow the user to seamlessly input their operands and place a hexadecimal point on whatever position they desire. For a visual of the Logic Finite-State Machine component, see References, *Figure-4*.

### *F. Top-file*

After all of the individual components were created, they were all combined together in a top-file. This file was called *FX_Calculator.vhd*. The inputs of this file include the two USB signals, which will be strictly for keyboard-to-microprocessor board communication. In addition, the clock and reset signals will be an input, as well, and will be distributed to all of the components that require these inputs to properly function. The first switch (SW0) on the microprocessor board will be utilized as an input to the circuit, and it will be tied to the finite-state machine, to signal the program when the user wants to see the results of the operations.

The main outputs of this circuit that allow for the 7-segment displays to function and display required values are the variables "leds" and "AN." The output variable "leds" is tied to the seven segments of each of the 7-segment displays, along with the dot associated with each of the displays. The output variable "AN" is tied to the driver of the eight 7-segment displays. Again, these variables come out of the serializer component, which is the component of the circuit responsible for displaying the right values on the correct displays.

In order to make sure that the correct keypresses are registered the program, the decoded keypress value (output of the decoder) is displayed by the onboard LEDs. This output of the top-file is labeled "led_keybrd." For a visual of the datapath, see References, *Figure-5*.

## III. Experimental Setup

During the course of putting together the project, piece by piece, every component was first tested after it was created, before being interconnected with other components.

First, the finite-state machine was built to be the driver of the registers. Then, the registers were connected to the finite-state machine. The outputs of the registers were then hooked up to the serializer component. From here, this portion of the project was simulated. Since the serializer component had only minor changes, no simulation for this component alone was required. The same idea holds for the registers as well. Thus, this circuit was tested by simulating various inputs (keypress decoder outputs), along with keypresses. The finite-state logic was examined carefully in order to ensure that the program was operating as expected. The registers were also closely examined to determine if they behaved as expected. Once this circuit was operating correctly, the arithmetic circuits component was built and simulated.

Within the arithmetic circuit, the testing was conducted by simulating various inputs (register outputs) and determining whether the outputs were displaying correct results. Since the addition, subtraction, and multiplication circuits came from the professor, these had no direct testing required. However, since the division circuit had to be built, this circuit was individually tested, in order to ensure that it was working properly. Once the entire arithmetic operations component was confirmed to be outputting proper results, this circuit was connected to the registers.

After testing all of the required components, the project top-file (*FX_calculator.vhd*) was then uploaded to the microprocessor board, and the calculator was tested with various inputs to determine if the program responds correctly to various cases.

## IV. Results

After the entire circuit was combined into the top-file (*FX_calculator.vhd*), the project as a whole was tested. Below are four example calculations made, and their obtained results:

$\boldsymbol{Addition}$: $90.23 + 1.4A0 = FF91.6D00$
$\boldsymbol{Subtraction}$: $0.2FB - 00.FB = FFFF.3460$
$\boldsymbol{Multiplication}$: $1A.90 \times 9.FF2 = F60.88C20$
$\boldsymbol{Division}$: $9.003 \div 3.003 = FFFFFFD.B$

For video demo, please click on the following link: https://www.youtube.com/watch?v=-FHj61elFNM

This demo goes through the addition and division example above, and also shows the other operation results on these operands.

## V. Conclusions

In the end, the project operates as expected. From a high-level standpoint, the end user will be able to input 16-bit, signed, fixed-point numbers and perform basic arithmetic operations on these numbers. The particular topic of fixed-point numbers and how to operate on these numbers was studied in the classroom. From the information and lectures provided by the professor on how these numbers need to be operated on, logic was created in VHDL to perform the required steps to complete each operation.

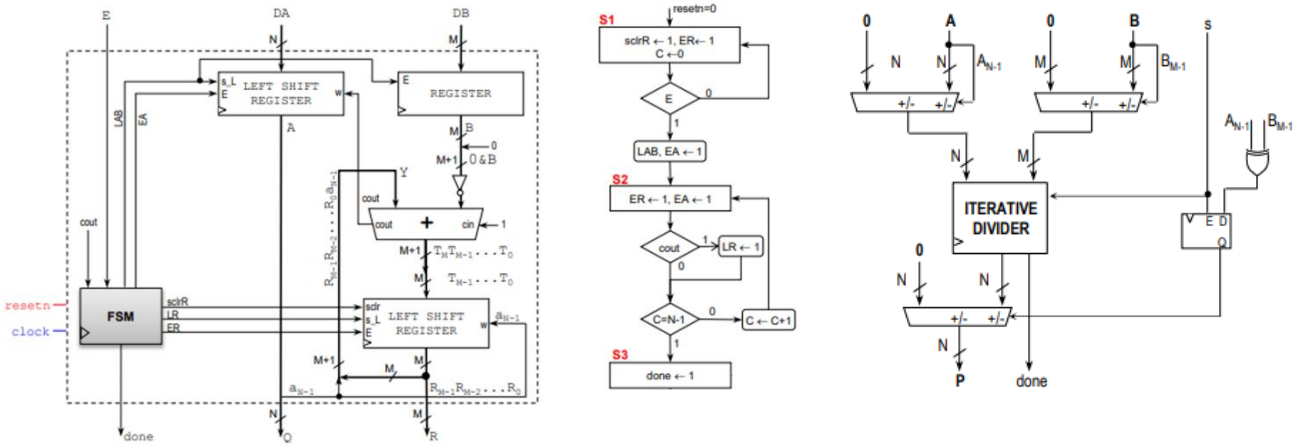
Although this program works and it outputs proper results, there still are some drawbacks to this project that need to be mentioned. For example, the program can only handle 16-bit numbers. Therefore, if the user wanted to perform an operation with the hexadecimal number F9 and they only input F9 into the first operand, the program will read this number as F900. Thus, the user must input 00F9, in order for the program to see the number as the user intended. Another issue that this program has deals with the user's ability to input a hexadecimal point into the operand. As it stands, if the user clicks on the dot key, the operand will keep incrementing by one, due to the nature of the code. Also, due to the keypress codes send by the keyboard component, at times, random keypresses show up and are accepted by the program as

random values. Thus, the user must be careful not to accidentally press an incorrect key.

While the mentioned issues above are not crucial to the overall operation of the project, they are still opportunity for future improvements. For example, the way the finite-state machine outputs the "dot" signal to the registers can be edited to account for cases where the hexadecimal point key may be pressed multiple times without changing the value of the respective register output. Also, further upgrades can be made to the keyboard decoder component and finite-state machine to specifically not accept any other keys besides for the ones regarding the hexadecimal numbers, hexadecimal point, and operation keys. Another upgrade could be made to the division logic; the dividend could be zero-extended by more than four fractional bits, in order to increase accuracy. Four was the limit for this project due to the fact that the dividend could not exceed 32-bits, as there are only eight 7-segment displays. Thus, another improvement that can be made is to upgrade the program to be able to handle larger numbers. For such a change, the largest hurdle will be to program the *arith_circuits.vhd* file to automatically display the dot on the required registers. With larger numbers, there are more places where the hexadecimal point could show up, meaning more cases need to be considered when outputting final results of operations.

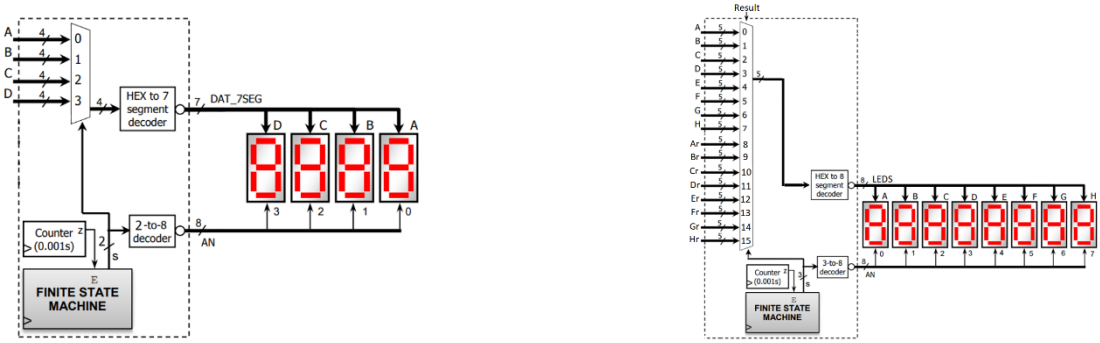
REFERENCES

*Figure 1. Signed Division Datapath circuits*




*1A. Iterative Divider (unsigned)* *1B. FSM* *1C. Signed Divider Circuit*

*Figure 2. Serializer Datapath circuits*




*2A. Original Serializer* *2B. Modified Serializer*

*Figure 3. Configured Keyboard*

*Figure 4. Logic Finite-State Machine*

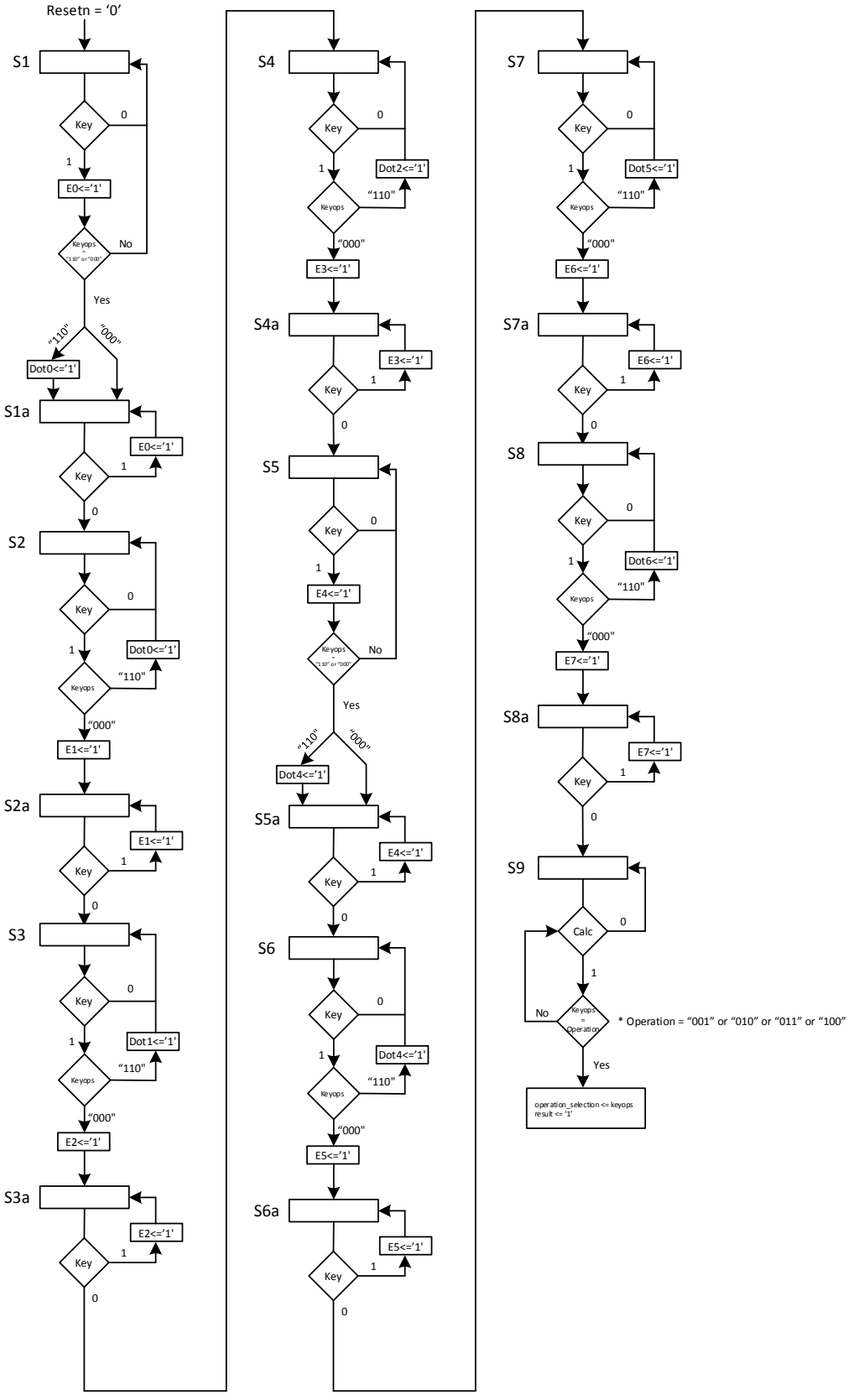


*Figure 5. Fixed-point Calculator Datapath*