microProcessor Dice Game
Designed in VIVADO & Implemented on NEXYS 4 DDR FPGA Board

Matthew Guirguis, Christopher Mathewson
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
mgguirguis@oakland.edu, cmathewson@oakland.edu

***Abstract*—The goal of the project is to design a microprocessor that can handle a D&D dice game that will receive instruction lines from a keyboard input and have data output through UART protocol.**

I. Introduction

The main concept of the project is based on a roleplaying system called Dungeons and Dragons 5th edition. The idea is that you roll a twenty-sided dice, use modifier values, and see if you can roll higher than a specific value. The player has an array of statistics that can be used to modify the rolls placed in memory, The goal of the mini-game is to roll higher than the enemy armor class in order to do damage in what is called the attack roll. Once the attack roll is decided to be higher or lower than the enemy armor class, the player is allowed to do damage. In the special case that the player rolls a perfect 20 out of 20, the game in instantly won and damage is doubled.

II. Methodology

The main approach to implementing the mini-game was to use a microprocessor to handle all of the calculations using an 8-bit instruction set. The assembly code is input and converted into machine code via a PS/2 keyboard input and an asynchronous PS/2 code to machine code instruction decoder. The player is then displayed predefined text depending on the situation that is determined by preset values based on the instruction used and the result obtained over the serial UART communication to a PC.

A. PS/2 Keyboard Input

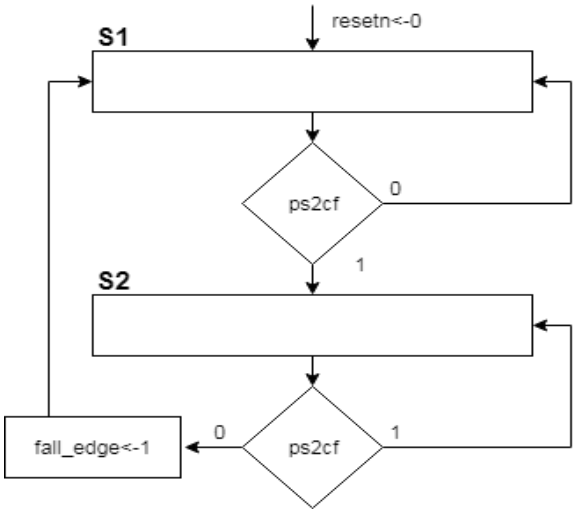

Figure 1. Falling Edge Detector

The keyboard functionality is intended to do two things: To show the command line being typed and to create a signal for the dice to be randomly rolled. Instructions will be decoded and immediately handled for the sake of simplicity rather than being stored in memory. A falling edge detector is used on the ps2c in conjunction with a modulo-10 counter to store the incoming serial PS/2 code inside an 10-bit right-shift register. An 8-bit right-shift register filter is used on the PS/2 clock to eliminate any noise, and data is received from the ps2d. Once the finite state machine has proceeds through 10 clock cycles, a done signal is sent out, and the PS/2 code is successfully stored inside a right-shift register. The first eight bits are then transferred to the assembler.

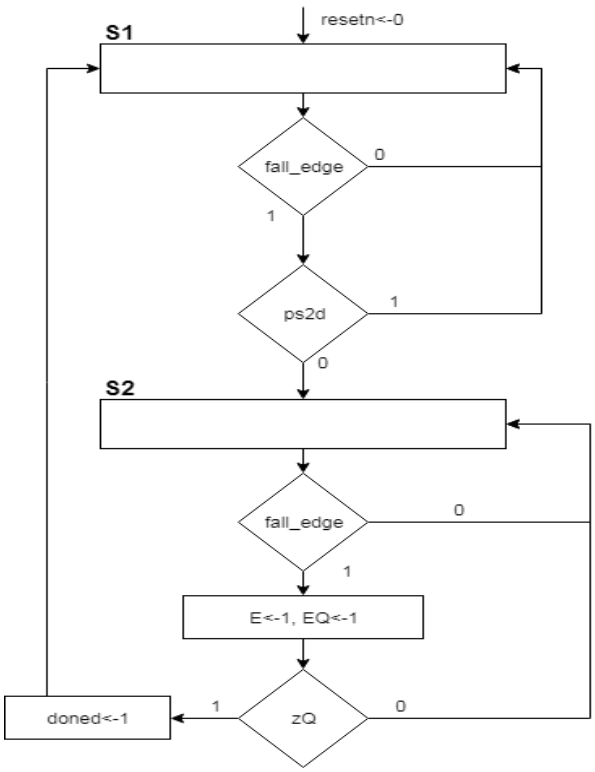

Figure 2. Main Keyboard FSM

B. Assembler

To store the assembly code, a 48-bit left-shift register is shifted by four to accumulate a total of 6 characters. Using the incoming PS/2 data, when the done signal is received from the PS/2 keyboard input. the data is checked to see if it is enter, x"5A". If it is, a start signal is sent to the microProcessor to start the instruction set. The 48-bit register data is sent through an asynchronous instruction decoder that directly translates from PS/2 code into machine code and sent to the microprocessor simultaneously. Once a ke-up code, x"F0" is found, the start signal is returned to zero, and the microprocessor is able to continue through its states.

C. microProcessor

1. Machine Code

The microprocessor is intended to handle 21 different instructions that get divided into three different categories. One set of instructions specifically handles writing data into memory. Another set of instructions handles doing the initial attack roll, which is the first part of the mini-game. These instructions include a dice roller, a register, an adder, and a comparator for flags. The last set handles the damage rolls, which uses different kinds of dice depending on the instruction given that is subtracted from a set value in memory.

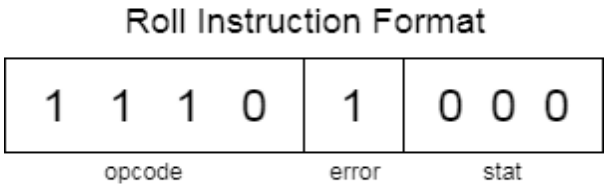

Figure. 3: Roll Instruction Format

In the roll instruction set, the first four bits are used the opcode, "1110", the 5th bit is used as an error check bit, and then last three bits are used as the memory address of the modifier used in the roll. The error bit was chosen due to the limitation of the 8-bits used, and it was the only bit that left every other instruction unmodified, effectively allowing for a no operation instruction.

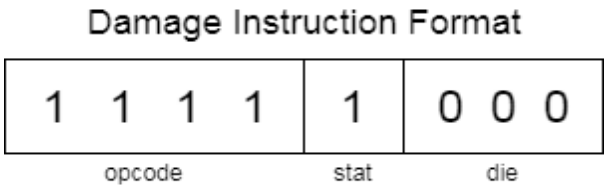

Figure 4. Damage Instruction format.

The damage instruction set is similar in that it has a four bit opcode, "1111." The 5th bit used for the address of either the strength or dexterity modifier, and the last three bits are used as the enable for the dice roller.

Figure 5. Stat Instruction Format

Lastly, the stat instruction only uses three bits for the opcode in order to increase the possible values stored in memory up to 31 in decimal by allowing a 5-bit operand to be used

2. Implementation

The initial state of the microprocessor waits for the start signal. Once received, the machine code is stored inside of an IR register, and the dice roller is enabled.

The dice roller is a set of 6 modulo counters set at different pulse timing. An enable decoder and multiplexer are then used to decide which die to use for the instruction.

An X and SUM register are then used to load the different modifiers from memory and add them together. In the case of an attack roll, before the data is sent to the output register, the data is compared with the enemy armor class value stored in memory to set the flag register. Being sent through a comparator, if the different of the attack roll and the armor class is at least zero, the hit flag is raised, allowing for the damage instruction to be used. On the chance that the innate roll is a perfect twenty, the hit flag and crit flag are immediately raised.

The damage instruction when received checks to see if the hit flag is raised. If it is, it proceeds to use the die chosen and add only the stat modifier to the roll and output it. If the crit flag is raised, the damage roll goes through an extra process to add another roll of the die chosen to the output register. After the instruction is finished, the flag register is then cleared.

In the final instruction set, data is stored directly from the operand in the machine code and stored in memory using the opcode as an address.

After any instruction is executed, the microprocessor then waits for the output FSM to finish loading the text to serial and then returns to idle.

D. Output Control

The three main purposes of the output control are to first, prior to any execution of assembly code, to directly convert PS/2 code to ASCII and transmit through serial; this is done by waiting for the done signal from the PS/2 keyboard control and serialize the 8-bit data that is stored in a register. The second purpose is to load the predefined text for the instruction used from emulated ROM. This is done by using an index and offset for each instruction. Once the text is fully loaded from the ROM, if the instruction and flag allow, the data in the output register from the microprocessor is converted from binary to ASCII and stored in a 16-bit register. The register is shifted eight bits as it is transmit through serial. The output control then transmits the proper new-line and carriage return ASCII commands, x”0A” and x”0D.” Once the serial data is sent, a done signal is sent back to the microprocessor, allowing it to return to idle and accept any new assembly lines.

1. UART TX

In order to transmit data successfully over a serial line, a constant high is sent through the UART to represent an idle state. Once the UART_TX is about to transfer 8-bits of data, a zero is sent as the start bit. Eight bits are transmit individually every 10416 clock cycles to create a 9600 baud rate that the serial port on the PC can understand. The transmission is ended with a stop bit that is high, and the serial signal returns to idle and awaits the next start bit.

II. Experimental Setup

The Nexys 4 DDR FPGA board is to be used alongside the PS/2 emulated keyboard input with a serial output using UART serial protocol. A PC will be used to read the outputs from the board. To display the serial communication between the board and the PC, a program called PuTTY is used to create the receiving baud rate and convert the ASCII characters to the letters and numbers that are shown on the screen. On the physical board, the lower 8 LEDs are used as the microprocessor’s output, and the higher LEDs are used to represent the machine code that is stored in the microprocessors IR register. A blue LED is used for the start signal, a green LED is used for the hit flag, and red LED is used for the crit flag.

III. Results and Conclusions

As a finished product, the board effectively executes each line of assembly code without fail. The data is transmit successfully over a serial port, and the flags and machine code are properly represented on the board as intended. The only unintended issue occuring is sometimes the enter key must be held down for a longer period than intended to successfully load the text from the ROM. If the key is not held down long enough, the microprocessor may not fully reset. In this case, the microprocessor gets stuck somewhere in the middle of its execution, and the enter key must be held down once again to successfully reset it back to idle.

During the development of the project, however, many different steps had to be taken to readjust and evaluate the design. During the first creation the microprocessor, the instruction format was changed several times to allow the best possible usage of each bit and to fulfill the necessary requirements of that instruction.

The largest issue was the discovery of the second PS/2 code sent after the upcode. Because of the second transmission, the initial input FSM design would not leave the start phase. To fix this, a secondary state had to accept the PS/2 code after the upcode and ignore it.

When tackling the development of the microprocessor, the concept of how exactly much work goes into a processor became a very arduous task. Each instruction had to be individually coded within the microprocessor and output control, making the finite state machines lengthy in VHDL code. All of the data conversions had to be hardcoded; when writing the predefined text, each character had to be individually defined in ASCII in the ROM, which also took an extended amount of time.

Overall, the project enlightened the team to how much work goes into just a simple microprocessor, let alone a fully operational CPU. The project proved to be a major stepping stone in gaining further processing hardware knowledge and development and to develop new techniques for the designing process.

IV. References

[1] Daniel Llamocca, *Oakland University. PS/2 Keyboard Emulation*

[2] Dungeons & Dragons 5e, Wizards of the Coast. *Dungeons and Dragons 5e Player's Handbook.*

[3] "UART, Serial Port, RS-232 Interface." *UART in VHDL and Verilog for an FPGA*, "www.nandland.com/vhdl/modules/module-uart-serial-portrs232.html"

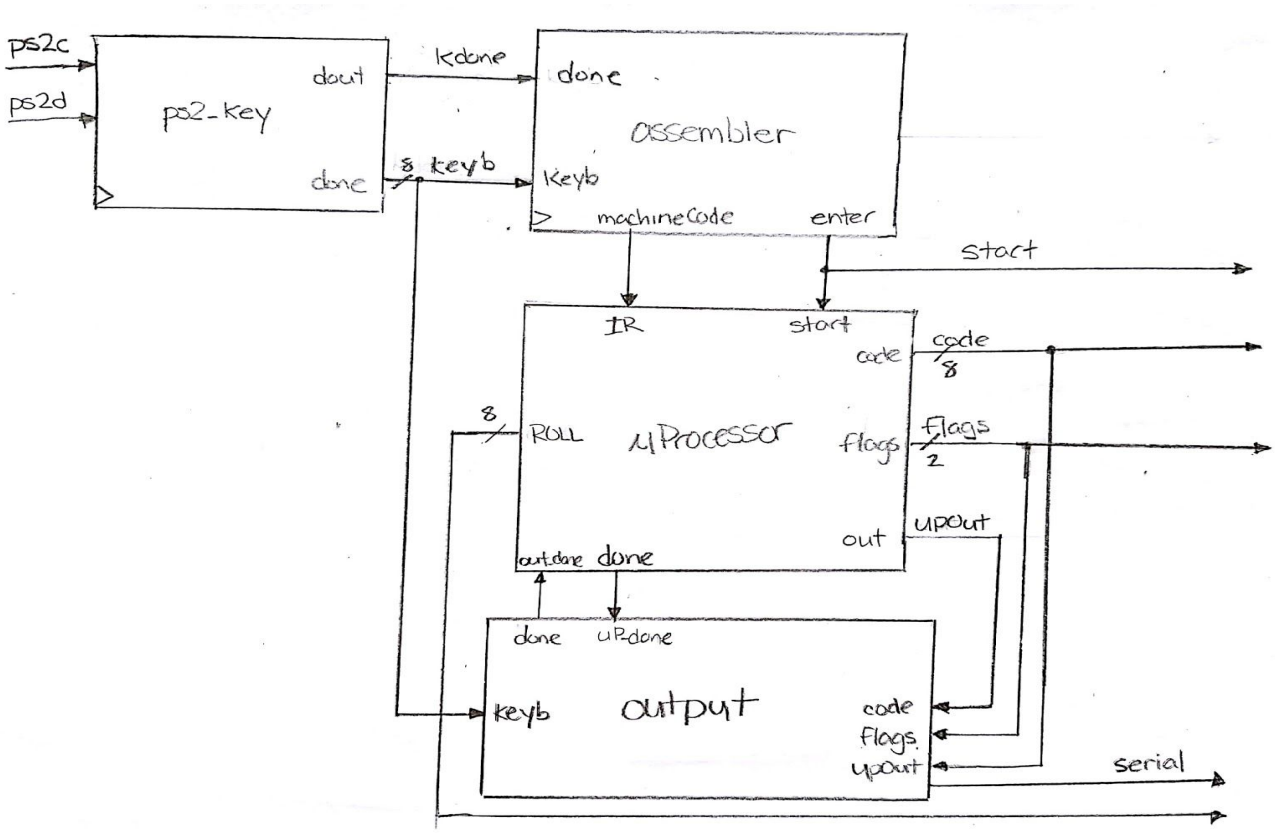

Figure 6. Top Level Design

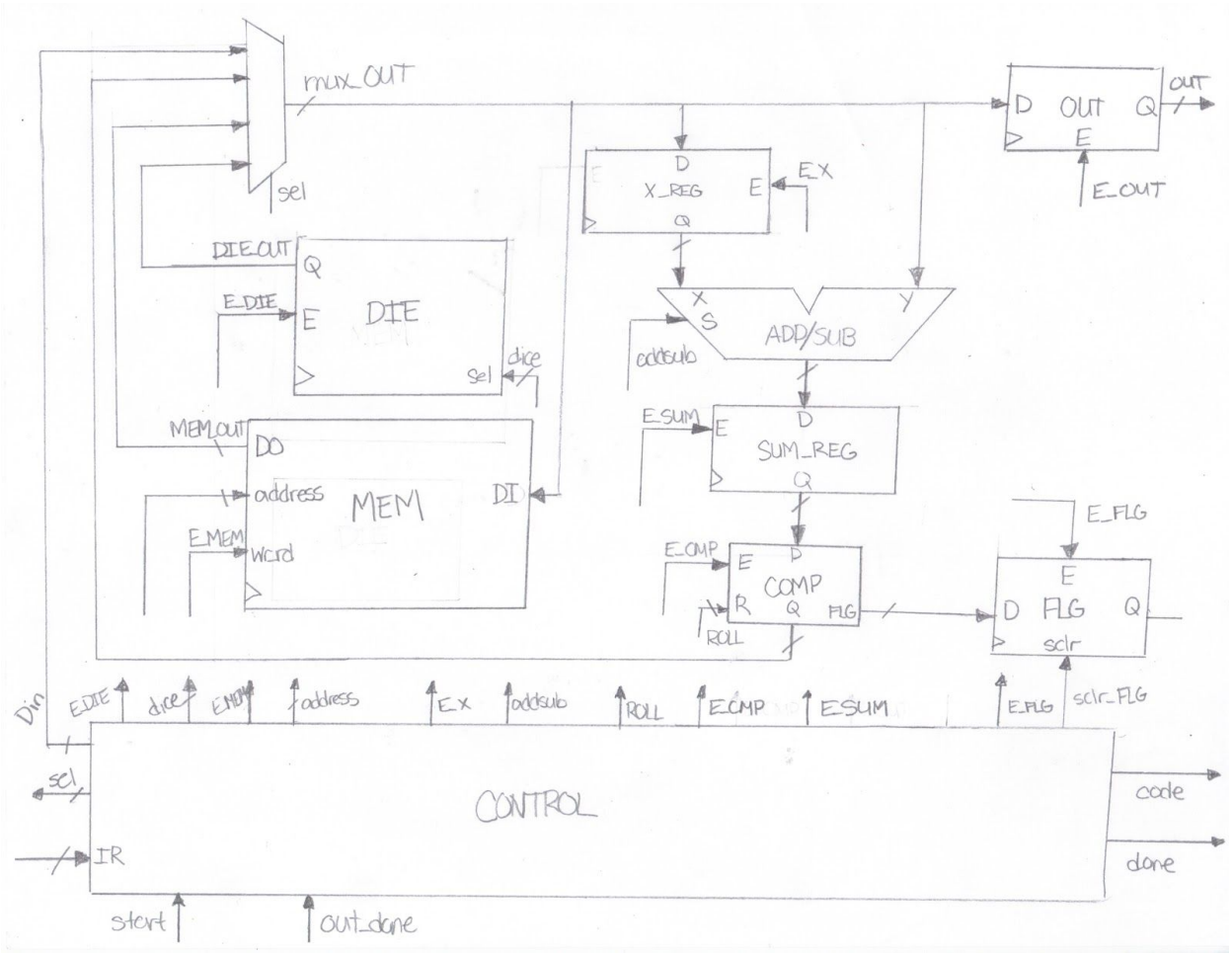

Figure 7. MicroProcessor Design

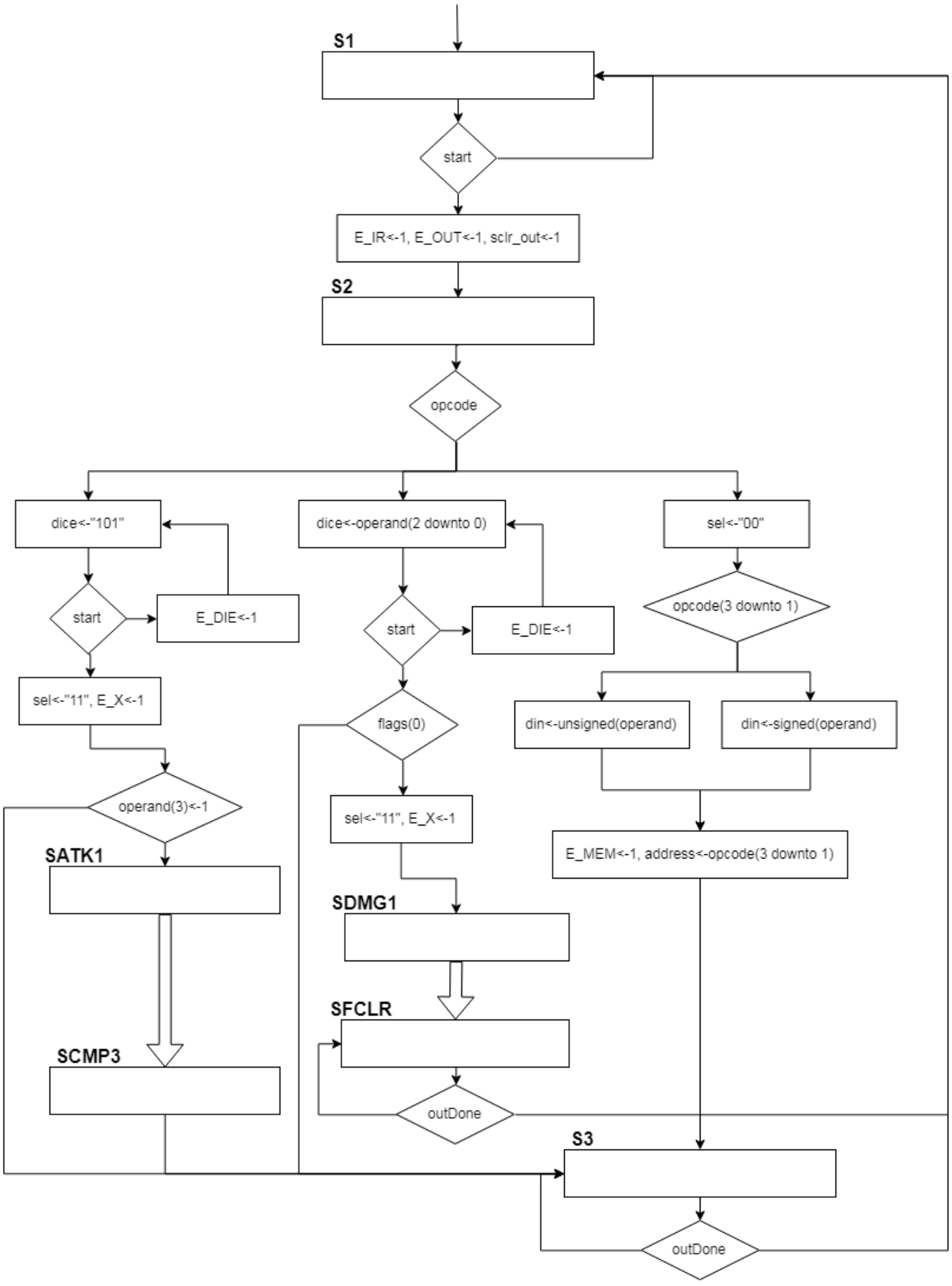

Figure 8. MicroProcessor FSM

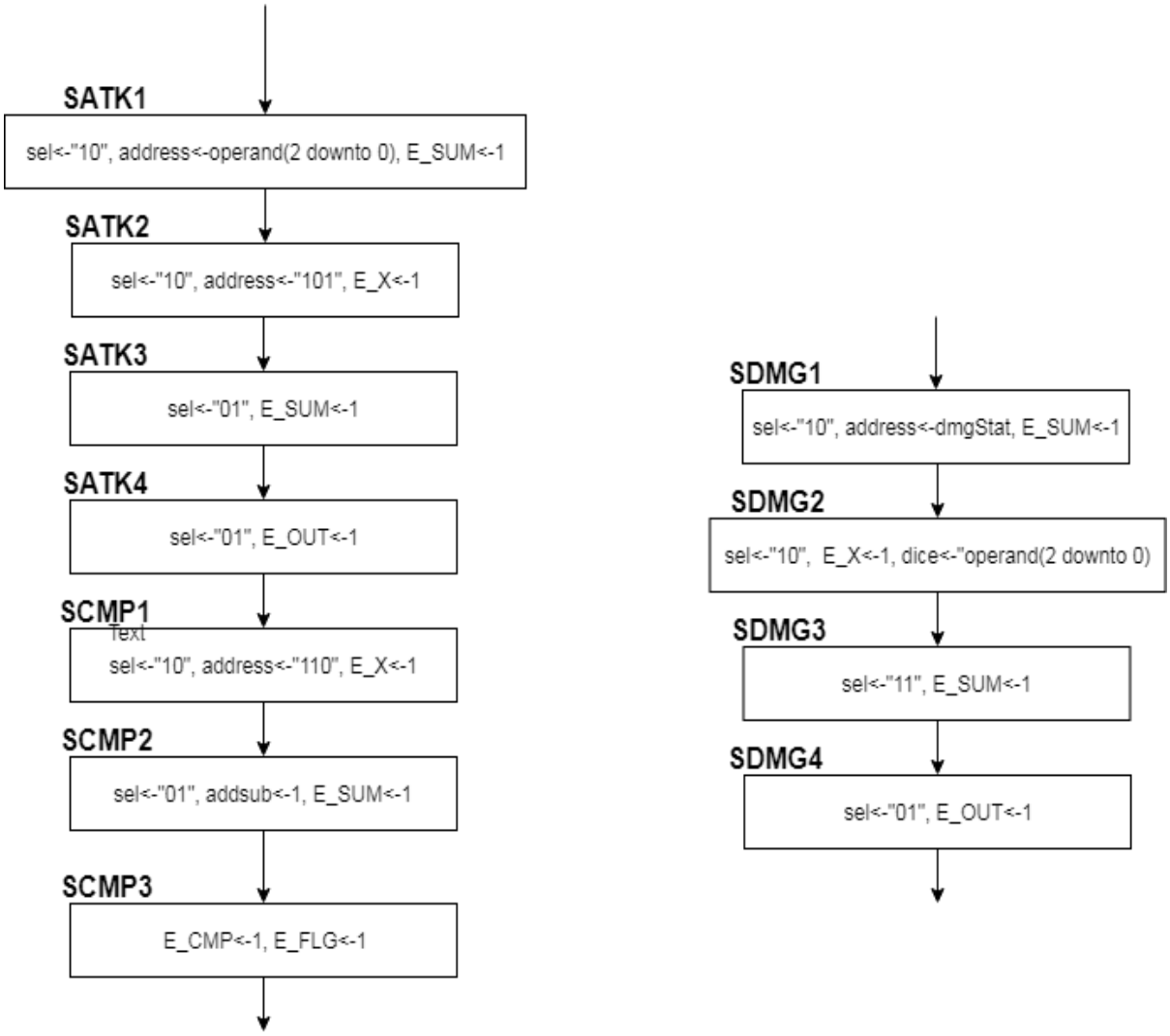

Figure 9. Instruction Roll and Damage FSM

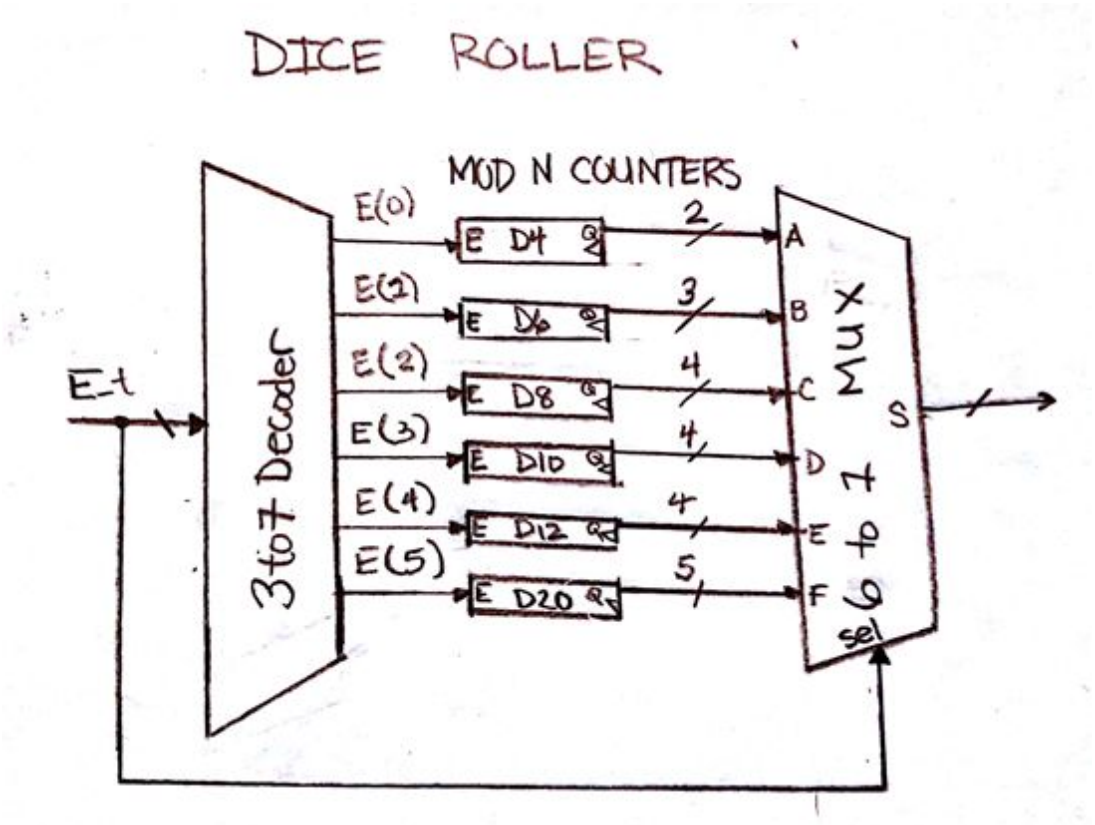

Figure 10. Dice Roller Design

Instruction	Machine Code
ROLLS	1110_0000
ROLLD	1110_0001
ROLLI	1110_0010
ROLLW	1110_0011
ROLLC	1110_0100

Figure 11. Roll Instruction List

Instruction	Machine Code
DMGDG	1111_1000
DMGSB	1111_1001
DMGLS	1111_0010
DMGGS	1111_0100
DMGAX	1111_0010
DMGGA	1111_0100
DMGWH	1111_0100
DMGCB	1111_1011
DMGLB	1111_1010

Figure 12. Damage Instruction List

Instruction	Machine Code
STR *op*	000 *op*
DEX *op*	001 *op*
INT *op*	010 *op*
WIS *op*	011 *op*
CHA *op*	100 *op*
PRF *op*	101 *op*
EAC *op*	110 *op*

Figure 13. Stat Instruction List

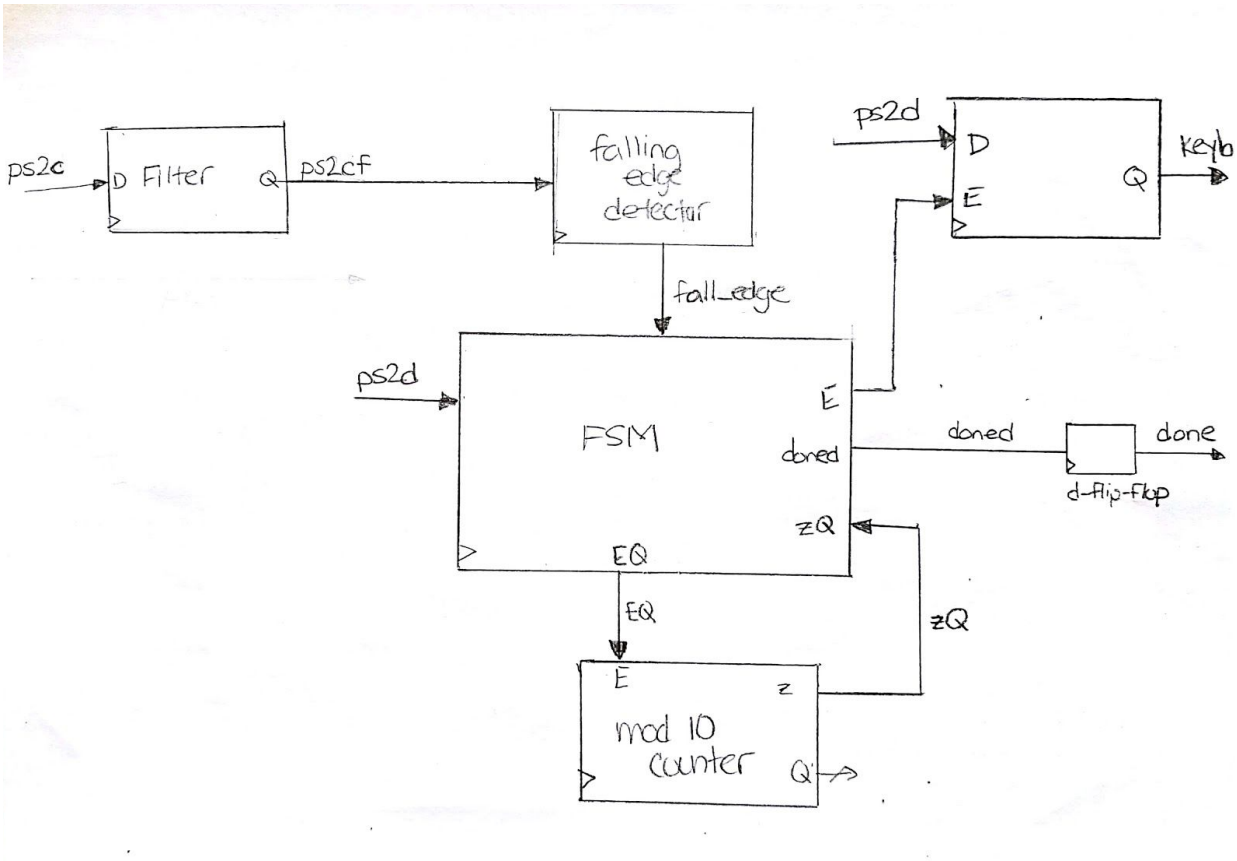

Figure 14. PS/2 Keyboard Control Design

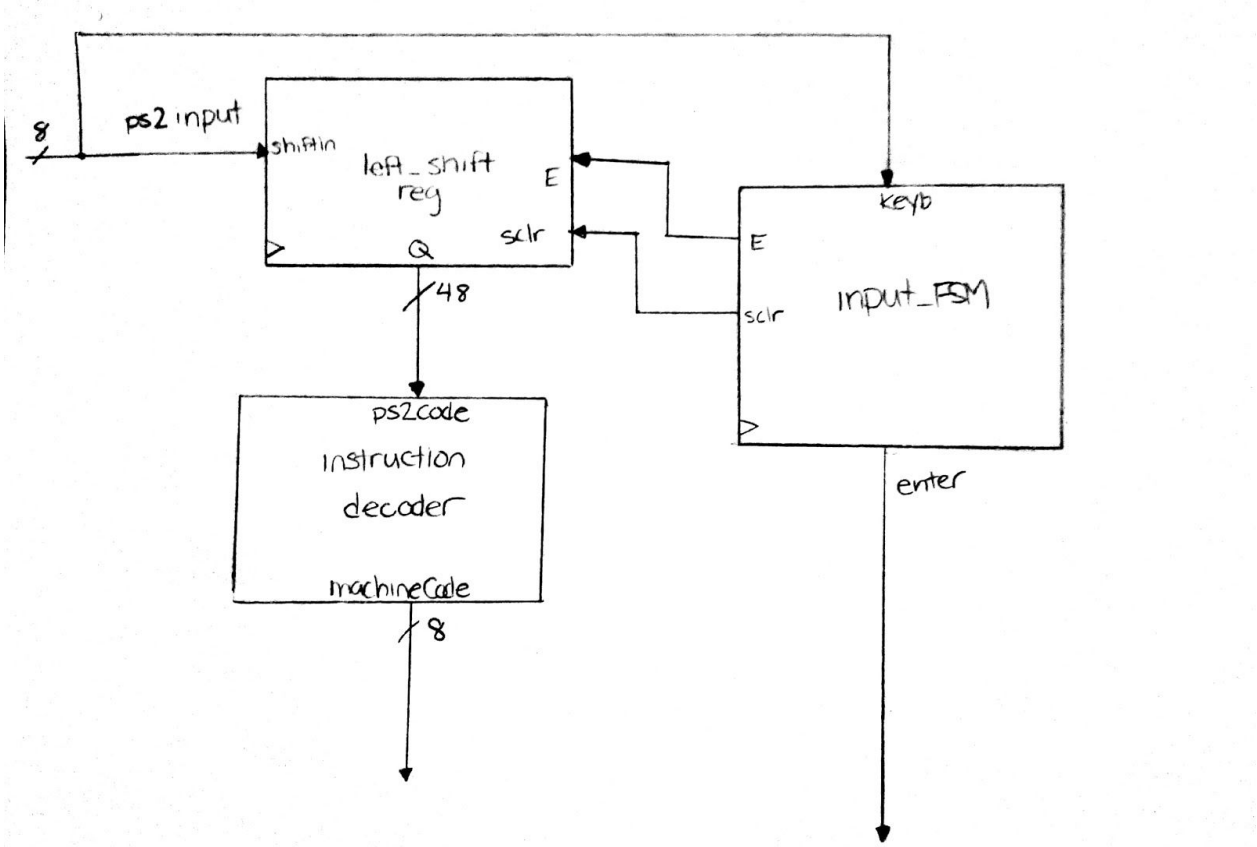

Figure 15. Assembler Design

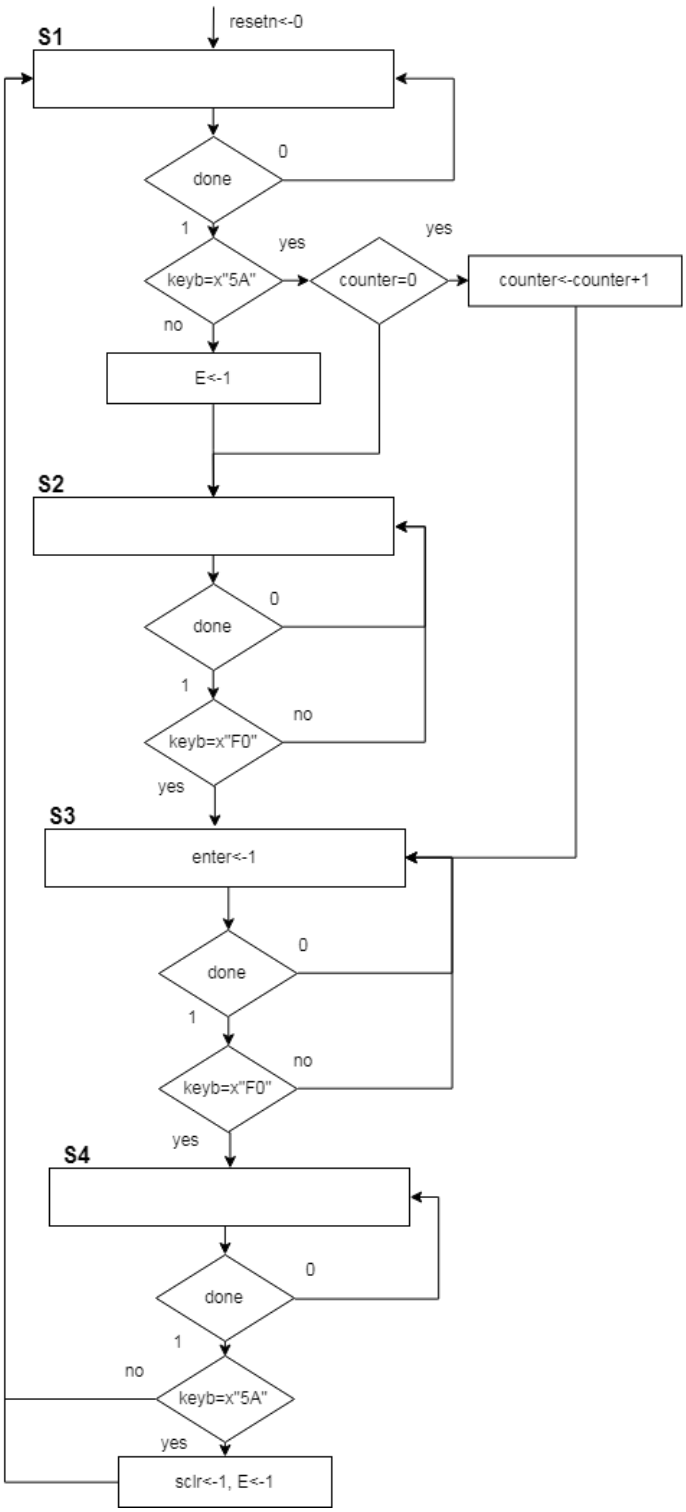

Figure 16. Assembler FSM

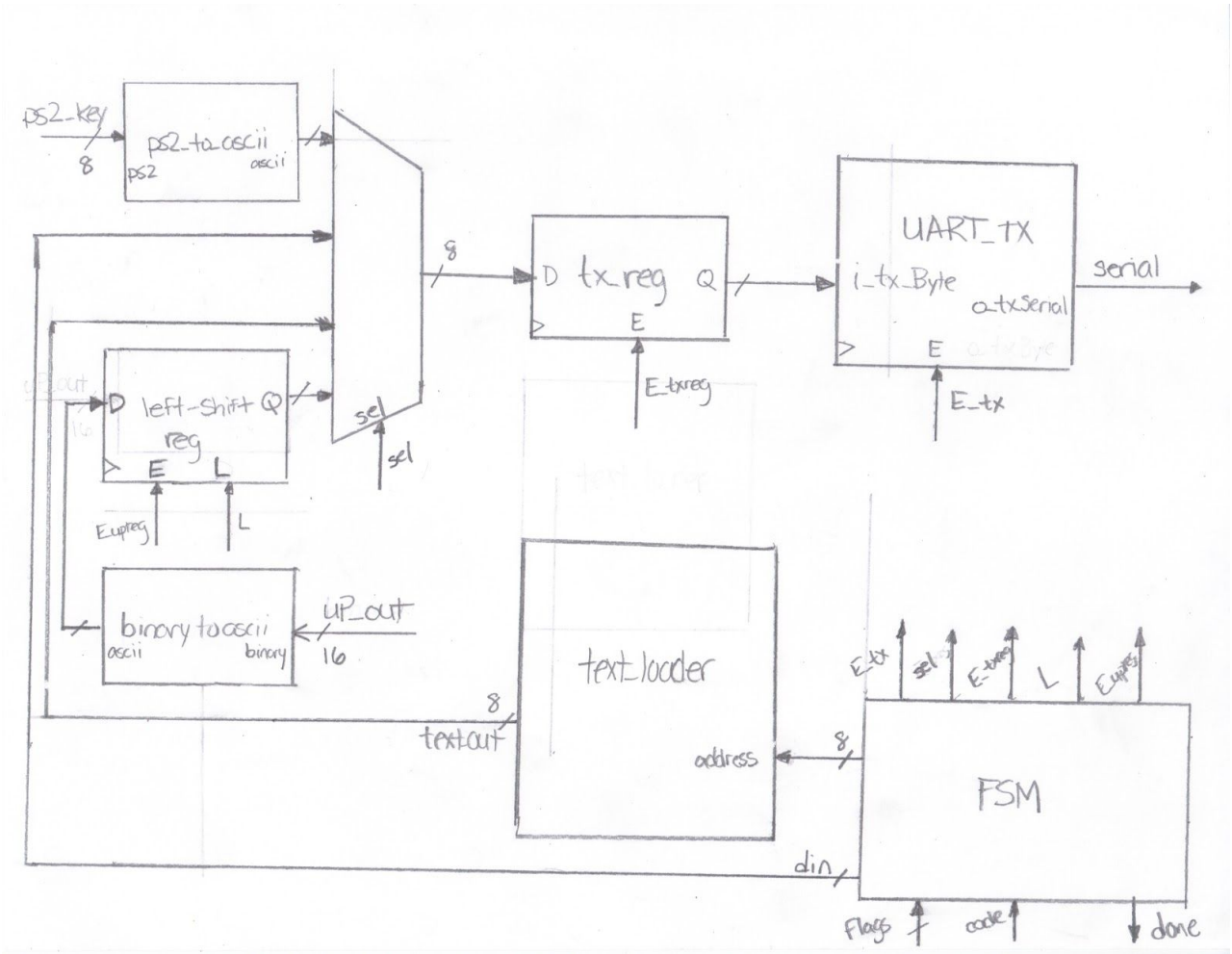

Figure 17. Output Control Design

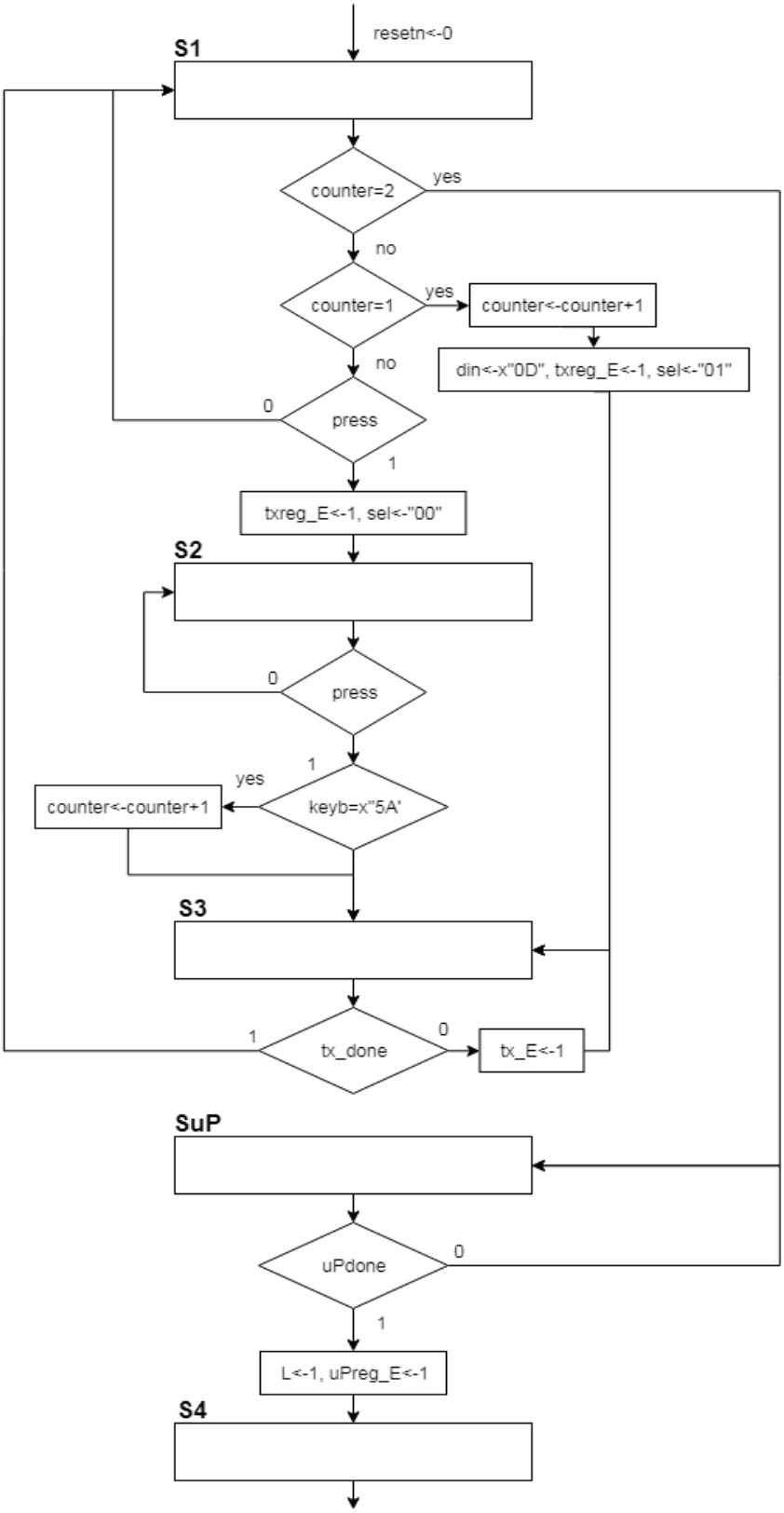

Figure 18. Output FSM Part 1

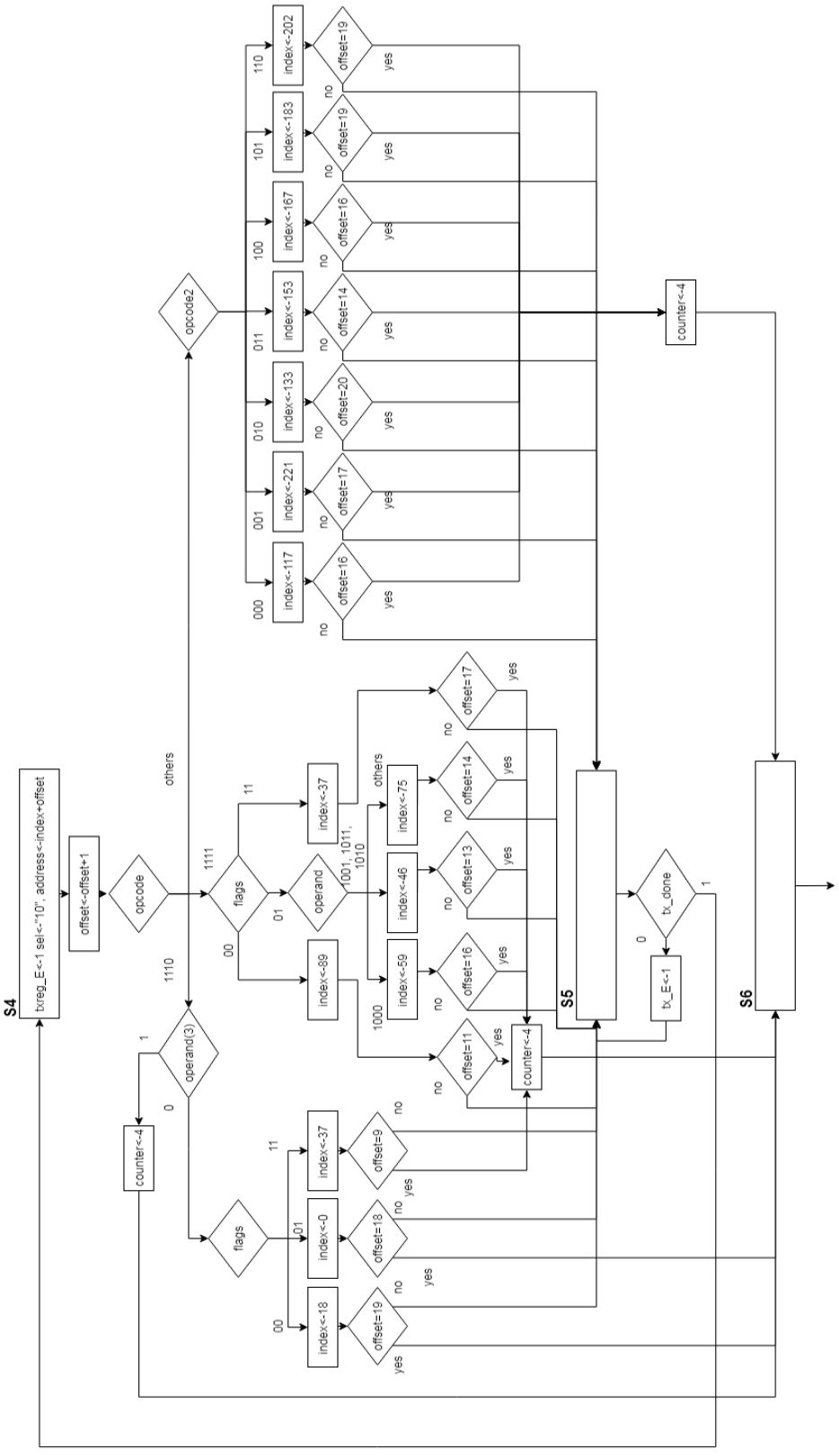

Figure 19. Predefined Text to Serial FSM

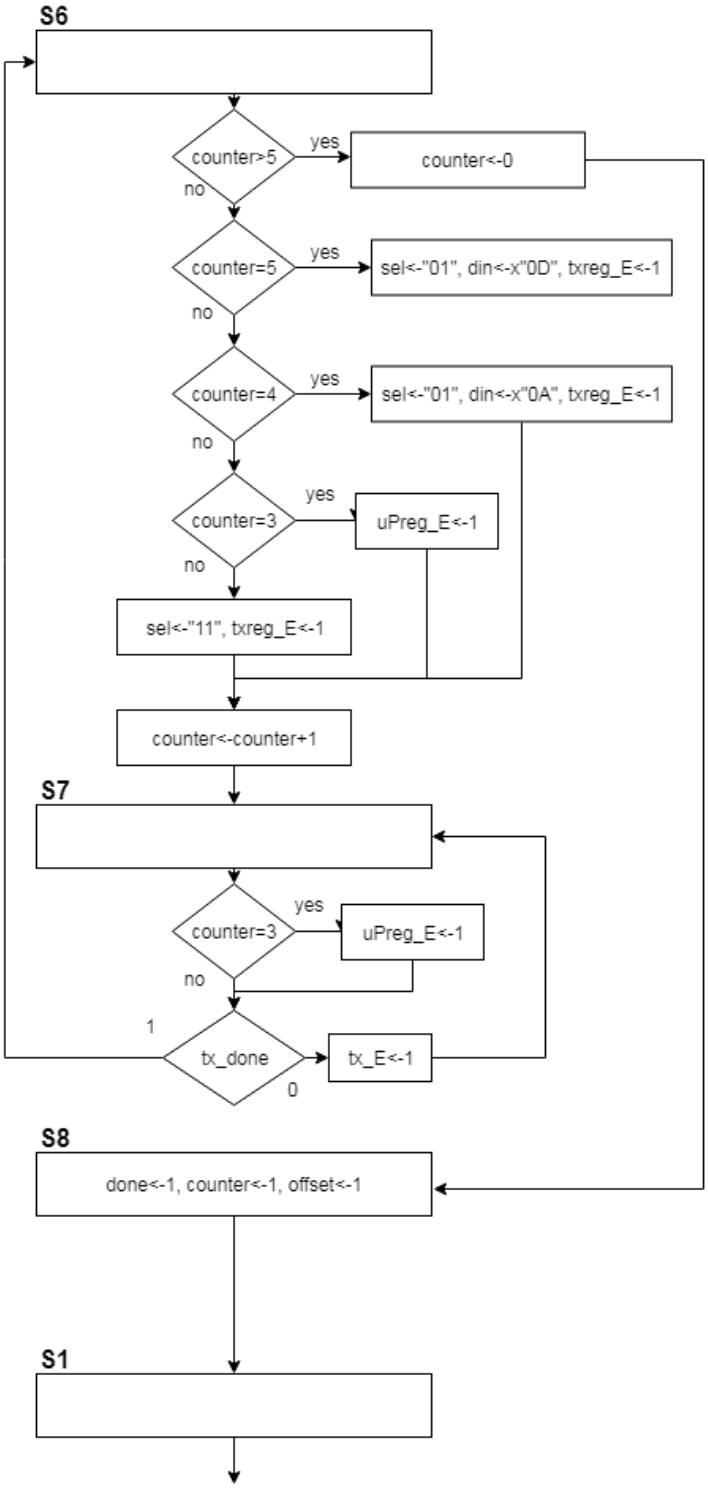

Figure 20. MicroProcessor Output to Serial FSM

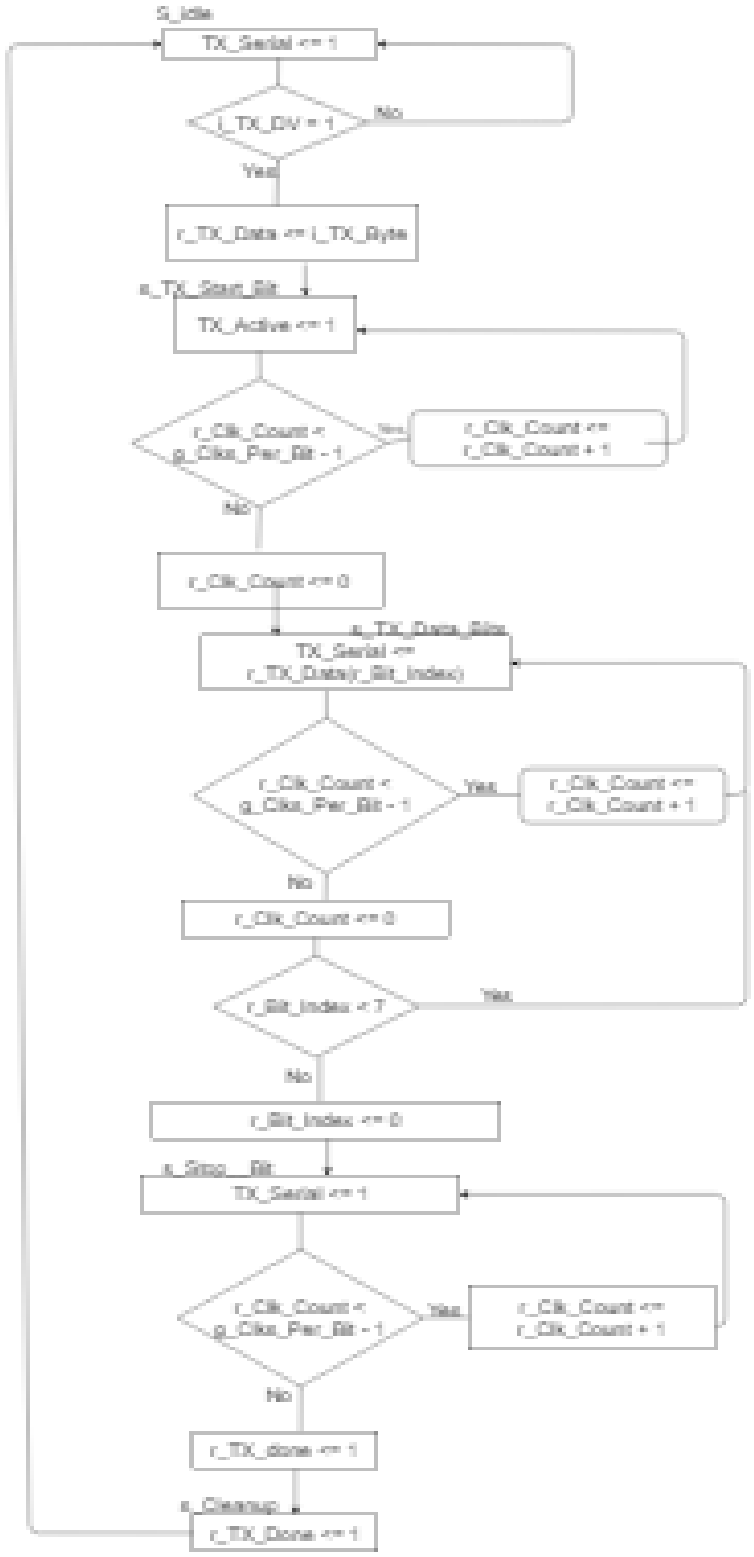

Figure 21. UART_TX FSM