EEPROM Programmer

Dorian Knight

Created: June 27th 2025
Updated: June 29th 2025

Contents

1 Project Overview **2**

2 Purpose **2**

3 Outcomes **2**
3.1 Hardware outcomes . . . 2
3.2 Firmware outcomes . . . 2
3.3 Software outcomes . . . 2
3.4 Project management outcomes . . . 3

4 Skills Used **3**

5 Design Requirements **3**
5.1 R1 . . . 3
5.2 R2 . . . 4
5.3 R3 . . . 4
5.4 R4 . . . 4
5.5 R5 . . . 4
5.6 R6 . . . 5

6 High Level Design **5**
6.1 Serial to Parallel Conversion . . . 5
6.2 Bidirectional Read and Write . . . 5
6.3 Manual User Debug . . . 5
6.4 Signal Multiplexing in EEPROM Motherboard . . . 5
6.5 Seven Segment Decoder . . . 6

7 Design Implementation Details **6**
7.1 Shift Register Design . . . 6
7.2 Debug Board Design . . . 10
7.3 Seven Segment Decoder Design . . . 11
7.3.1 Combinational logic derivation overview . . . 11
7.3.2 A-segment combinational logic . . . 12
7.3.3 B-segment combinational logic . . . 13
7.3.4 C-segment combinational logic . . . 14
7.3.5 D-segment combinational logic . . . 16
7.3.6 E-segment combinational logic . . . 17
7.3.7 F-segment combinational logic . . . 18
7.3.8 G-segment combinational logic . . . 20
7.3.9 Combinational logic simulation . . . 21
7.4 Motherboard Design . . . 21

8 Bill of Materials Formation and Considerations **23**

1 Project Overview

The EEPROM programmer is a custom-made hardware device with software support designed to interface a computer with the AT28C64B electrically erasable programmable read-only memory (EEPROM) [Link to data sheet] . For development and testing, custom firmware was made for the ATmega328p chip to drive the programmer. The EEPROM programmer also comes with bespoke software allowing the user to control the EEPROM programming from their computer.

2 Purpose

The purpose of this project was to venture far outside my comfort zone and tackle a problem I had no idea how to solve. I conceived this project idea while learning about the Pickit series programmers after asking questions about how they worked and why I couldn't program a PIC the same way I programmed an Arduino.

Deciding to make a custom EEPROM programmer allowed me to gain the following skills:

- Gain printed circuit board (PCB) design experience
- Introduction to firmware and C language drivers
- Increase fluency in reading and understanding datasheets
- Using datasheet information to compose a bill of materials
- Use digital marketplace tools (digikey search) to select components that met requirement specifications but also didn't break the bank
- Learn what it is like to manage a project – even if it was just a project with one developer who was also the manager

3 Outcomes

I worked through the hardware-software co-design process to create a custom EEPROM programmer PCB along with a custom software application to control the programming of a 64 KB EEPROM.

3.1 Hardware outcomes

- Created bidirectional serial to parallel interface allowing the Arduino's serial data I/Os to interface with the EEPROM's parallel addressing and data requirements
- Derived combinational digital logic through Karnaugh mapping to create a binary to seven-segment decoder
- Designed switching mechanism allowing the user to control the programmer through MCU firmware or user-driven debug board

3.2 Firmware outcomes

- Created custom C driver running on Arduino hardware to read from and write to an EEPROM using custom hardware bidirectional serial to parallel interface
- Supported error checking during EEPROM flashing transcription accuracy
- Communicates with custom software application over USB cable using UART
- Able to guarantee 99% accuracy when programming up to XX KHz

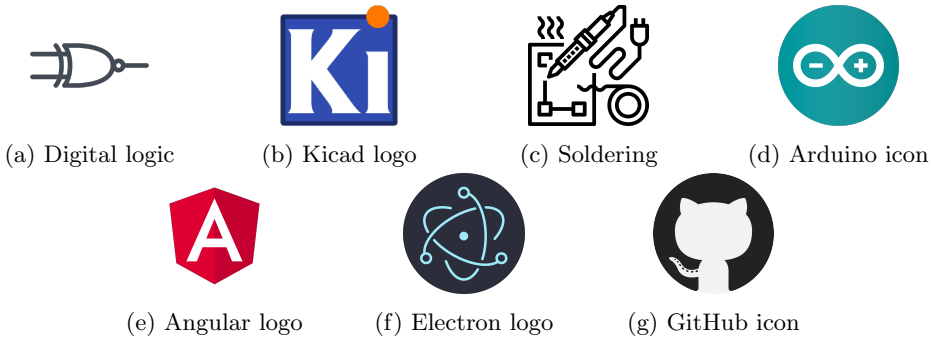
3.3 Software outcomes

- Used Angular framework with Electron to create a sleek graphical user interface (GUI) to interface with the MCU on the EEPROM Programmer

3.4 Project management outcomes

- Leveraged modular design to minimize development risks and parallelize development
- Strategically used Digikey component quantity price breaks to reduce component budget by 35%

4 Skills Used



(a) Digital logic (b) Kicad logo (c) Soldering (d) Arduino icon

(e) Angular logo (f) Electron logo (g) GitHub icon

Figure 1: Skills used constructing EEPROM programmer

Digital logic was used to derive the combinational logic required to translate binary to seven-segment display control signals. **Kicad** was used to design and visualize the physical hardware of the EEPROM Programmer. **Soldering** was used to assemble all the components once I received the manufactured PCBs from JLCPCB. **Arduino** was used to drive the individual circuit components to accomplish the end goals of flashing data to the EEPROM. Arduino was also used to interface with the programmer software application such that the user could directly program the EEPROM from their computer. **Angular** was used to design the programming application and **Electron** was used to package that application making the software available for users on Windows, Linux and MacOS. **GitHub** was used to version control the Angular software application.

5 Design Requirements

To ensure project feasibility I drafted requirements during the project planning stage to set the initial project scope.

The requirements were as follows:

- R1: Must be controllable through microcontroller unit (MCU) or manual hardware debugger
- R2: Should use LED indicators to show the status of pins of interest (address, data, control signals)
- R3: All PCBs must have a power LED indicator that is illuminated when the board is powered.
- R4: Must minimize port interface size with MCU to increase the number of commercial MCUs that could be used to drive the EEPROM programmer hardware.
- R5: Must be programmable using software application on user's computer.
- R6: Should perform error checking/correction during or after flashing information to the EEPROM.

Here is how the EEPROM programmer project adhers to the requirements above:

5.1 R1

To allow for MCU or direct user control, a switching mechanism was used to control multiplexing between the shift register and debug board signals. When the MCU control was selected, only the shift register and Arduino signals were let through to directly control the EEPROM. When debug mode was selected the multiplexers restricted the flow of information from the shift register and Arduino and allowed the signals from the debug board to pass through to the EEPROM.

5.2 R2

LEDs were used throughout the design on the shift register and the motherboard. The shift register LEDs indicate the status of each data flip-flop while the LEDs on the motherboard indicate the "post multiplexed" signal is on the address, data and control signal lines. Additional seven-segment display hardware was designed to humanize the data readout.

5.3 R3

An LED indicator was placed on all PCB boards to indicate whether or not the board was powered.

5.4 R4

Through the implementation of a bidirectional shift register, I was able to reduce the pin count from 28 pins to 12 pins greatly expanding the types of MCUs that could be used with my EEPROM programmer hardware.

Without serial to parallel converter the MCU needs 28 pins (2 power + 26 GPIO) to drive the following signals:

- 8 data
- 13 address
- 3 control
- 1 power
- 1 ground
- 1 USB TX
- 1 USB RX

With the serial to parallel converter the MCU only needs 12 pins (2 power + 10 GPIO) to drive the following signals:

- 1 data in (from the shift register for error correction)
- 1 data out (from Arduino to shift register)
- 1 address out (from Arduino to shift register)
- 3 control signals
- 1 load signal (used to reverse the serial-¿parallel interface into a parallel-¿serial interface)
- 1 clock signal (used to sequentially advance the data through the shift register
- 1 Power
- 1 Ground
- 1 USB TX
- 1 USB RX

5.5 R5

Angular application packaged for desktop using electron is used to interface the user's computer with the onboard MCU.

5.6 R6

Every time a byte is written from the shift register into the EEPROM it is read back into the shift register then shifted out 1 bit at a time into the Arduino to be checked against the original byte that was sent out. If the received byte does not match the sent byte, the address is rewritten and checked again. This ensures that each address has the intended value.

By following these requirements, I was able to keep scope creep to a minimum and deliver a product that holds true to the initial inception of the project's goals.

6 High Level Design

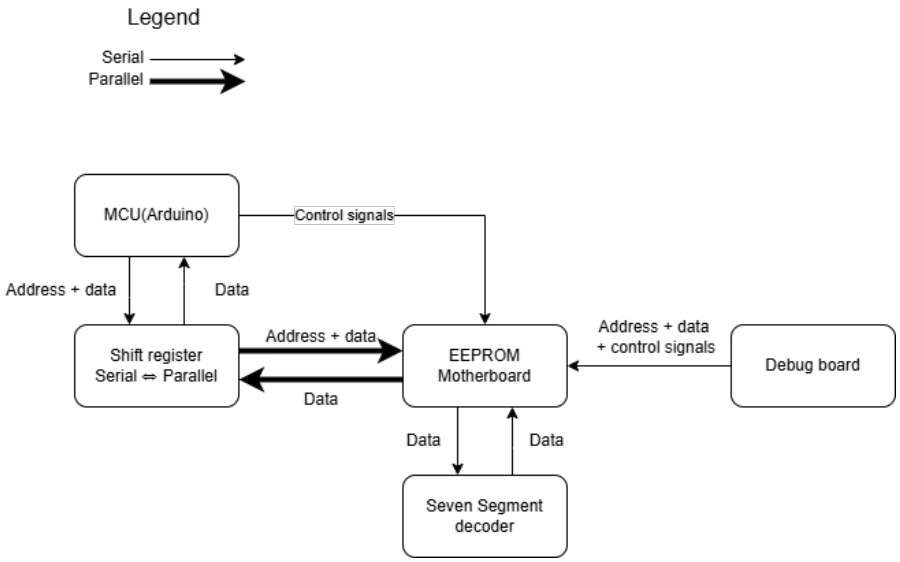


Figure 2: EEPROM programmer hardware high level block diagram

6.1 Serial to Parallel Conversion

Based on the system requirements the MCU interface is serial while the EEPROM interface is parallel. To rectify this difference, a custom bidirectional shift register is used such that the Arduino can pipe in the data one bit at a time and when sufficient data has been written, the Arduino can signal the EEPROM to read or write that specific address.

6.2 Bidirectional Read and Write

While data can be piped in one at a time into the shift register, data can also be loaded in parallel from the EEPROM into the shift register then shifted out to the Arduino controller one bit at a time. This is done to support error correction as the Arduino can compare what it intended to write vs what actually got written.

6.3 Manual User Debug

A custom debug board was designed to interface with the EEPROM Motherboard. When put into debug mode, the debug board dip switches and push buttons can be used to directly program the EEPROM.

6.4 Signal Multiplexing in EEPROM Motherboard

Both the shift register and debug board signals feed into multiplexer inputs and depending on the state of the MCU/debug switch either the shift register signals or debug board signals are let through to the EEPROM pins.

6.5 Seven Segment Decoder

A custom PCB was made to convert a 4-bit binary number into the seven signals required to drive a seven-segment display. This was done to prove my digital logic design capabilities. The PCB takes the 4 bits as an input and uses combinational logic derived through Karnaugh mapping to output the seven signals required to drive a seven-segment display. Two 7SD PCBs are used where each PCB drives one seven-segment display. One display represents the lower 4 bits in an address and the other represents the higher 4 bits. Together they represent every bit from 0x00 to 0xFF (0-255).

7 Design Implementation Details

7.1 Shift Register Design

The shift register design was created to resolve the serial to parallel disconnect between the MCU and EEPROM. R4 (from section 5.4) mandates that the required MCU GPIOs be kept to a minimum in order to maximize the list of compatible MCUs that could be used to drive the EEPROM programmer hardware. To reduce the number of required GPIOs, address and data bits are sent out one bit at a time serially instead of sending all bits at once in parallel. This reduces the number of address and data GPIO pins down from 21 (8 data + 13 address) to 2 (1 data + 1 address).

While the MCU interface was changed to support serial data flow, the EEPROM still required its data and address bits to be delivered in parallel. Looking at section 15 in the EEPROM datasheet we can see that the write waveform requires all 13 address bits and 8 data bits when the chip enable and write enable goes low.

15. AC Write Waveforms

15.1 $\overline{WE}$ Controlled

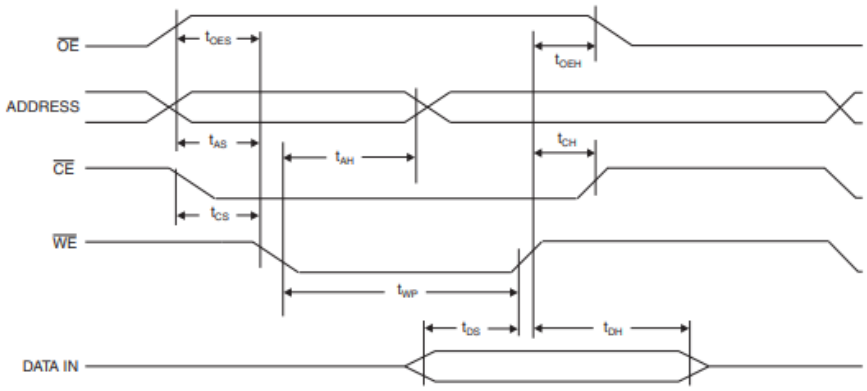


Figure 3: EEPROM writen enable controlled write waveform

Looking at the physical pin layout in figure 4 we can see that the 13 address bits and 8 data bits are all connected to separate legs of the IC requiring the driver to control the state of each chip leg at the time of writing.

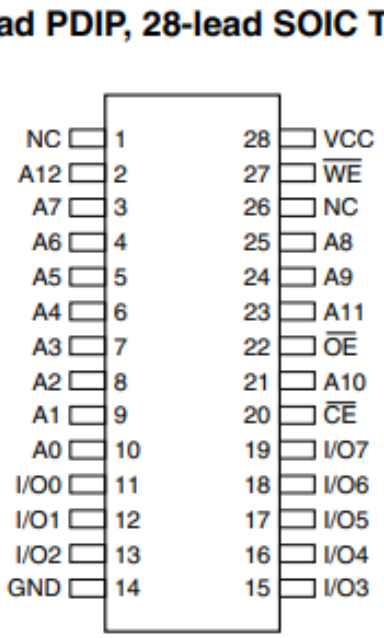


Figure 4: EEPROM IC pin layout

We only have two available pins (address and data) so the MCU can't control the state of each individual leg at once. The solution to this disconnect was a serial to parallel shift register where the MCU could push in the address and data one bit at a time and the shift register would output all 13 address and 8 data bits at once when the entire bit stream had been output by the MCU. This will allow the two MCU GPIOs to control the 21 address and data legs of the EEPROM.

Unfortunately, the solution gets more complicated. R6 (from section 5.6) mandates that the MCU perform error checking processes either during or after programming to ensure transcription accuracy. To do this, the shift register must be bidirectional and allow the parallel address and data bits to be serially shifted out of the shift register back to the MCU for verification.

To support both R4 (from section 5.4) and R6 (from section 5.6), a custom bidirectional shift register PCB was developed. This bidirectional shift register supported MCU serial data and address inputs while also supporting parallel loading from the EEPROM data lines into the individual flip-flops when a "load" control signal is provided from the MCU. After loading, the data, now in the individual flip-flops that make up the shift register could be shifted out to the MCU one bit at a time.

The entire shift register design can be represented using the following functional block diagram in figure 5:

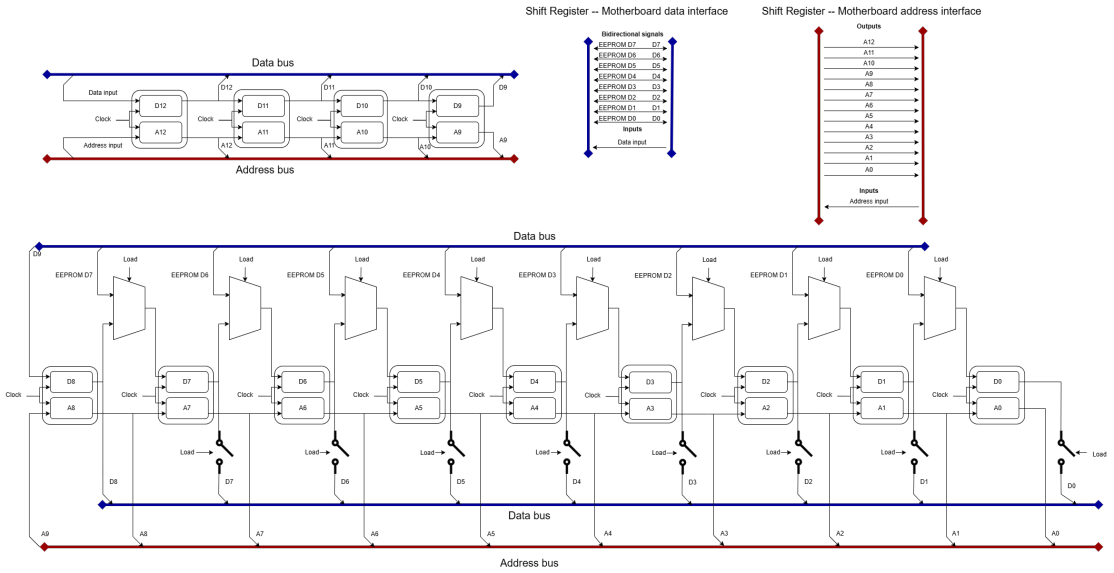


Figure 5: Shift register functional block diagram

The fundamental hardware unit used to assemble the bidirectional shift register was a dual level data flip-flop shown in figure 6.

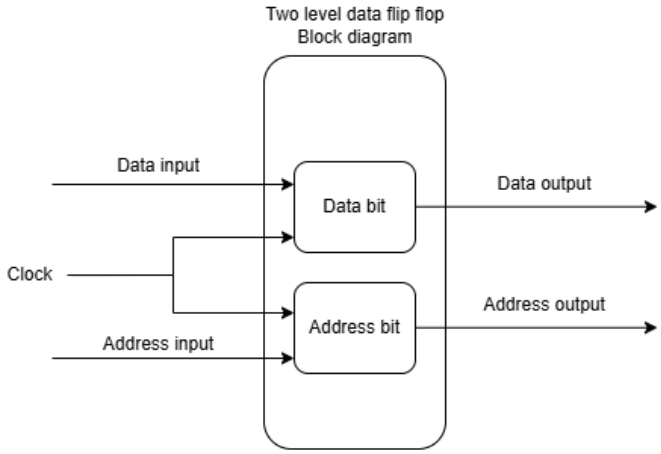


Figure 6: Dual level data flip-flop

This data flip-flop supports the storage of two individual bits through separate inputs and separate clock lines. The individual flip-flops were linked together such that the data output of one became the data input of the other shown in figure 7.

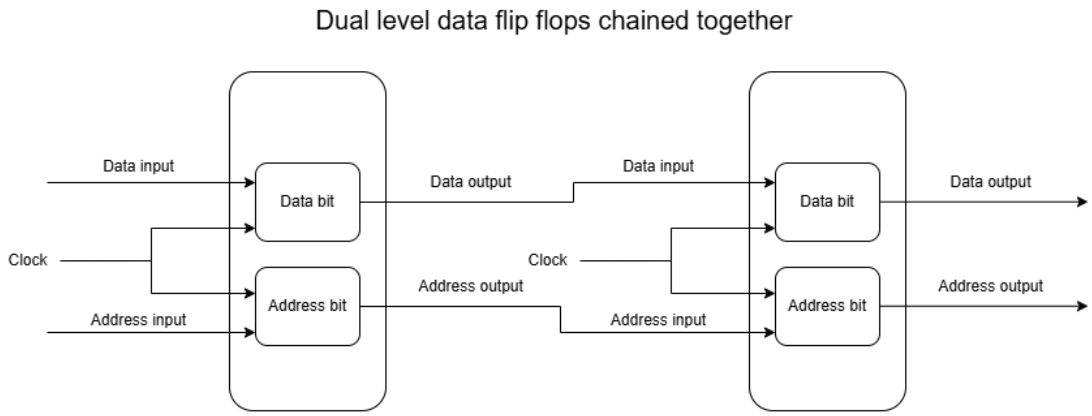


Figure 7: Two dual level data flip-flops linked together

All flip-flops share the same clock line to advance both the data and address bits at the same time, reducing the number of required clock lines from 2 to 1.

Thirteen dual level flip-flops were used to design the bidirectional shift register. This allowed each address bit to be represented in a flip-flop. However, since there were thirteen address bits, there were also thirteen data bits, of which only bits 0-7 were used by the EEPROM therefore only data bits 0-7 were connected through to the motherboard and into the EEPROM.

To support parallel loading, the data input lines were fed from a multiplexer shown in figure 8 which took the previous flip-flop's data and the EEPROM's data as inputs. The multiplexer was controlled by a "load" signal provided by the MCU requiring an additional GPIO pin.

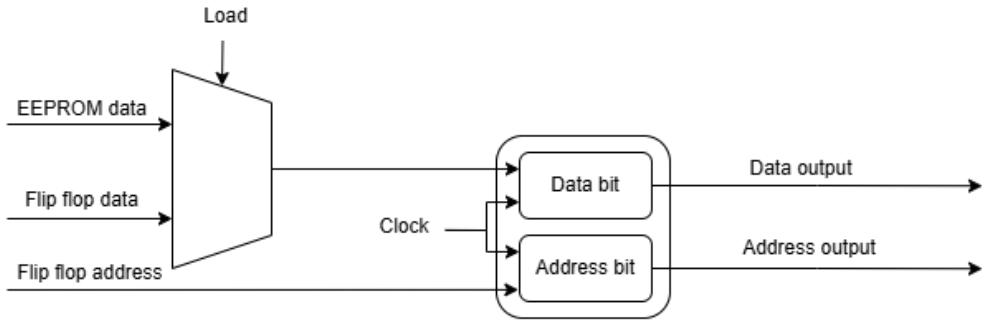


Figure 8: Flip-flop data input multiplexed to accept EEPROM data or shift register data

Because the data pins on the EEPROM are bidirectional, the data output from the flip-flops and the EEPROM data fed into the multiplexer actually come from the same PCB trace. To ensure the flip-flop data output doesn't pollute the signal trace when trying to load data from the EEPROM, a switch was used to turn off the flip-flop output as seen in figure 9.

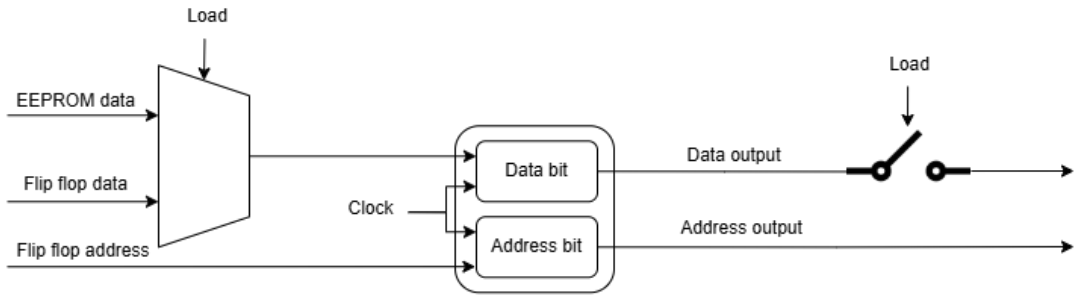


Figure 9: Flip-flop output controlled through load signal

The switch is controlled by the same load signal to reduce the required MCU GPIOs. When the load signal is active, the switch opens and the multiplexer switches to forward the EEPROM data. On the next clock signal, the EEPROM data bit will be stored in the shift register such that it can eventually be shifted out and read by the MCU.

Data bits 0-7 are the only outputs that need to be switched as data bits 8-12 are kept internally on the shift register board and are not piped forward to the motherboard.

7.2 Debug Board Design

The debug board was designed to satisfy R1 (from section 5.1). The manual hardware debugger is a combination of dip switches and buttons the user can use to physically control all the necessary signals to program individual addresses on the EEPROM. The main use of this board is for debugging purposes; to test if the correct byte is written on a specific address and if not, being able to rewrite that address with a new byte.

The debug board design can be represented using this functional block diagram in figure 10:

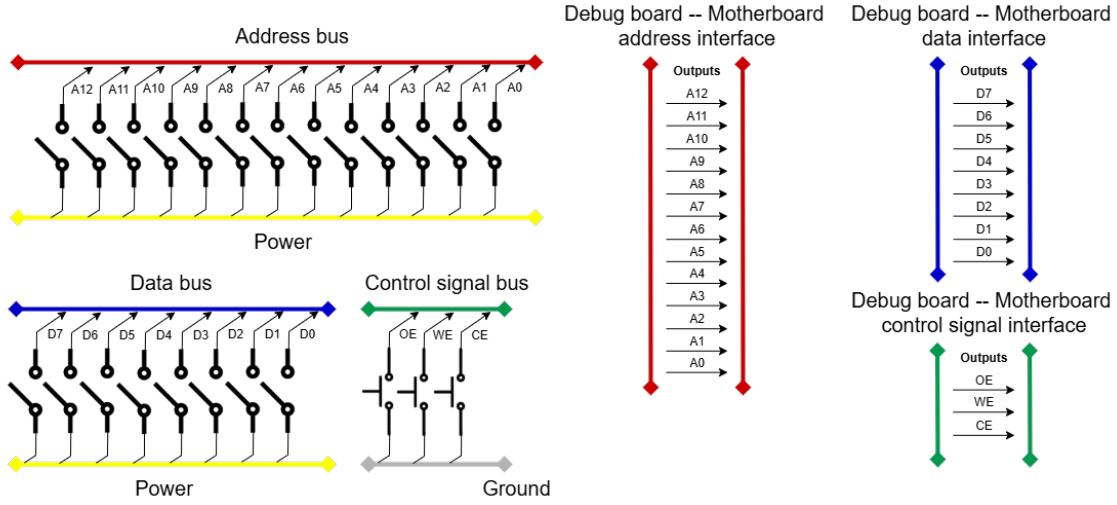


Figure 10: Debug board functioanl block diagram

Because the control signals are active low, the buttons are pulled high to Vcc and then the push button pulls the respective control signal back to ground when pressed. The remainder of the address and data signals are all pulled low to ground and the dip switches pull the signal line high to Vcc.

These signals are all piped through to the motherboard so that they can be multiplexed with the shift register signals, the output of which is piped through directly to the EEPROM.

7.3 Seven Segment Decoder Design

The seven-segment decoder was designed to satisfy R2 (from section 5.2). While the motherboard does have indicator LEDs showing the status of all address, data, and control signal lines, I'm more comfortable decoding hexadecimal than raw binary especially when the numbers get sufficiently large. To support a hexadecimal read out of the binary data at a specified address, I synthesized digital logic through Karnaugh mapping to take in a 4-bit binary number and output the required control signals to illuminate the hexadecimal representation of said number on a seven-segment display.

There are many valid ways to solve this problem. One would be to program an EEPROM where each address corresponded to a binary number and the data at that address mapped to the on/off status of each segment in a seven-segment display. It also was entirely possible to reuse one of my seven-segment decoder ICs from my time in undergrad but I decided to create my own combinational logic to showcase my digital logic synthesis skills.

The seven-segment decoder PCB can be represented using the following functional block diagram in figure 11.

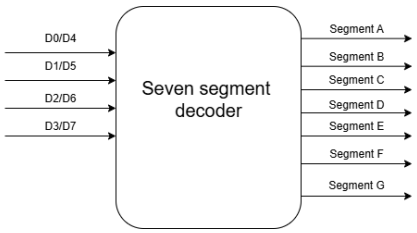


Figure 11: Seven segment decoder functional block diagram

There are two seven-segment decoder PCBs, one for bits D0-D3 and bits D4-D7. The corresponding segment outputs are piped to the two seven-segment displays housed on the motherboard. The two seven-segment displays represent the lowest four significant bits for D0-D3 and the four highest significant bits for D4-D7.

7.3.1 Combinational logic derivation overview

To derive the combinational logic, I had to create seven separate circuits which all took the four bits as input and outputted a zero or one for whether its respective segment should be turned off or on respectively.

Table 1 shows the Boolean state of each segment for every possible input from the motherboard.

Decimal representation	Input(binary)	Output (segments A-G)	7SD output
0	0000	1111110	0
1	0001	0110000	1
2	0010	1101101	2
3	0011	1111001	3
4	0100	0110011	4
5	0101	1011011	5
6	0110	1011111	6
7	0111	1110000	7
8	1000	1111111	8
9	1001	1110011	9
10	1010	1111101	a
11	1011	0011111	b
12	1100	1001110	C
13	1101	0111101	d
14	1110	1001111	E
15	1111	1000111	F

Table 1: 7SD combinational logic overview

7.3.2 A-segment combinational logic

Table 2 is a subset of table 1 displaying the individual Boolean state of the A-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	A segment output
0	0000	1
1	0001	0
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	1
7	0111	1
8	1000	1
9	1001	1
10	1010	1
11	1011	0
12	1100	1
13	1101	0
14	1110	1
15	1111	1

Table 2: A-segment Boolean state based on 4-bit input

Writing table 2 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 12.

D3D2/D1D0	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	1	0	1	1
10	1	1	0	1

$$\overline{D2}\,\overline{D0} + D3\overline{D0} + D3\overline{D2}\,\overline{D1} + \overline{D3}D2D0 + \overline{D3}D1 + D2D1$$

$$D3\overline{D0} + D3\overline{D2}\,\overline{D1} + \overline{D3}D2D0 + \overline{D3}D1 + D2D1 + \overline{D2}\,\overline{D0}$$

$$D3(\overline{D0} + \overline{D2}\,\overline{D1}) + \overline{D3}(D2D0 + D1) + D2D1 + \overline{D2}\,\overline{D0}$$

Figure 12: A-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 13.

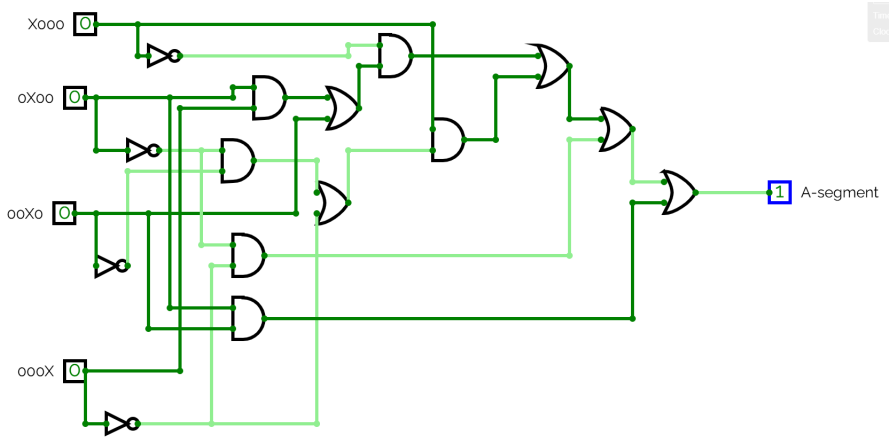


Figure 13: A-segment combinational logic

7.3.3 B-segment combinational logic

Table 3 is a subset of table 1 displaying the individual Boolean state of the B-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	B segment output
0	0000	1
1	0001	1
2	0010	1
3	0011	1
4	0100	1
5	0101	0
6	0110	0
7	0111	1
8	1000	1
9	1001	1
10	1010	1
11	1011	0
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Table 3: B-segment Boolean state based on 4-bit input

Writing table 3 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 14.

D3D2/D1D0	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	0	1	0	0
10	1	1	0	1

$$\overline{D3}\,\overline{D1}\,\overline{D0} + \overline{D2}\,\overline{D1} + \overline{D2}\,\overline{D0} + D3\overline{D1}D0 + \overline{D3}D1D0$$

$$\overline{D3}\,\overline{D1}\,\overline{D0} + \overline{D3}D1D0 + \overline{D2}\,\overline{D1} + \overline{D2}\,\overline{D0} + D3\overline{D1}D0$$

$$\overline{D3}(\overline{D1}\,\overline{D0} + D1D0) + \overline{D2}(\overline{D1} + \overline{D0}) + D3\overline{D1}D0$$

Figure 14: B-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 15.

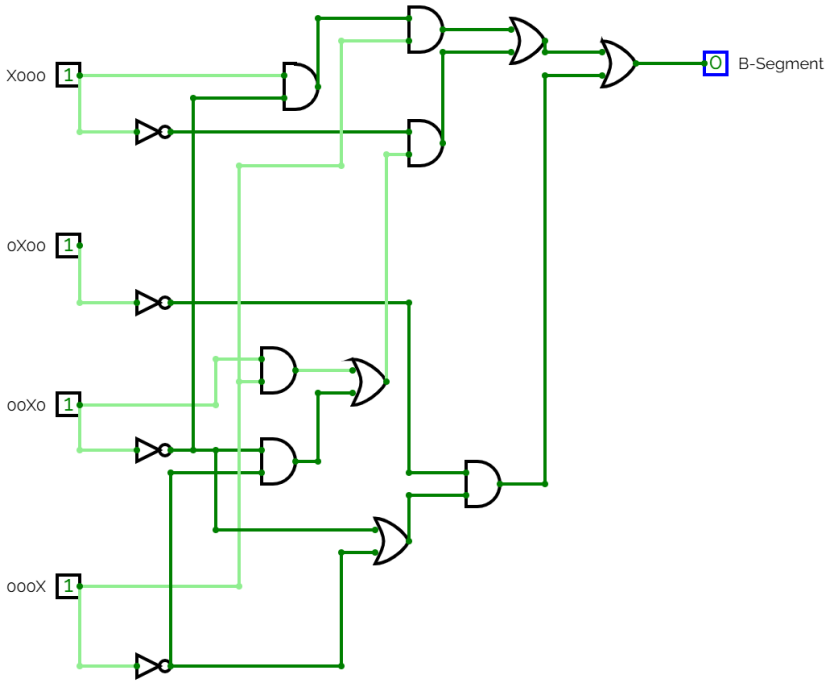


Figure 15: B-segment combinational logic

7.3.4 C-segment combinational logic

Table 4 is a subset of table 1 displaying the individual Boolean state of the C-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	C segment output
0	0000	1
1	0001	1
2	0010	0
3	0011	1
4	0100	1
5	0101	1
6	0110	1
7	0111	1
8	1000	1
9	1001	1
10	1010	1
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Table 4: C-segment Boolean state based on 4-bit input

Writing table 4 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 16.

D3D2/D1D0	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	0	1	0	0
10	1	1	1	1

$$\overline{D3}\,\overline{D1} + \overline{D3}D2 + D3\overline{D2} + \overline{D3}D0 + \overline{D1}D0$$

$$\overline{D3}\,\overline{D1} + \overline{D3}D2 + \overline{D3}D0 + D3\overline{D2} + \overline{D1}D0$$

$$\overline{D3}(\overline{D1} + D2 + D0) + D3\overline{D2} + \overline{D1}D0$$

Figure 16: C-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 17.

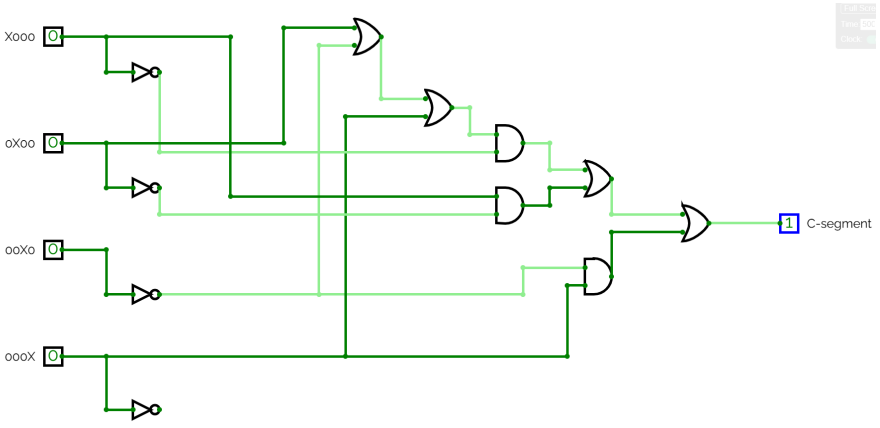


Figure 17: C-segment combinational logic

7.3.5 D-segment combinational logic

Table 5 is a subset of table 1 displaying the individual Boolean state of the D-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	D segment output
0	0000	1
1	0001	0
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10	1010	1
11	1011	1
12	1100	1
13	1101	1
14	1110	1
15	1111	0

Table 5: D-segment Boolean state based on 4-bit input

Writing table 5 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 18.

D3D2/D1D0	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	1	1	0	1
10	1	0	1	1

$$\overline{D2}\,\overline{D0} + D3D2\overline{D1} + D2\overline{D1}D0 + \overline{D2}D1 + D1\overline{D0}$$

$$\overline{D2}D1 + D1\overline{D0} + D3D2\overline{D1} + D2\overline{D1}D0 + \overline{D2}\,\overline{D0}$$

$$D1(\overline{D2} + \overline{D0}) + D2\overline{D1}(D3 + D0) + \overline{D2}\,\overline{D0}$$

Figure 18: D-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 19.

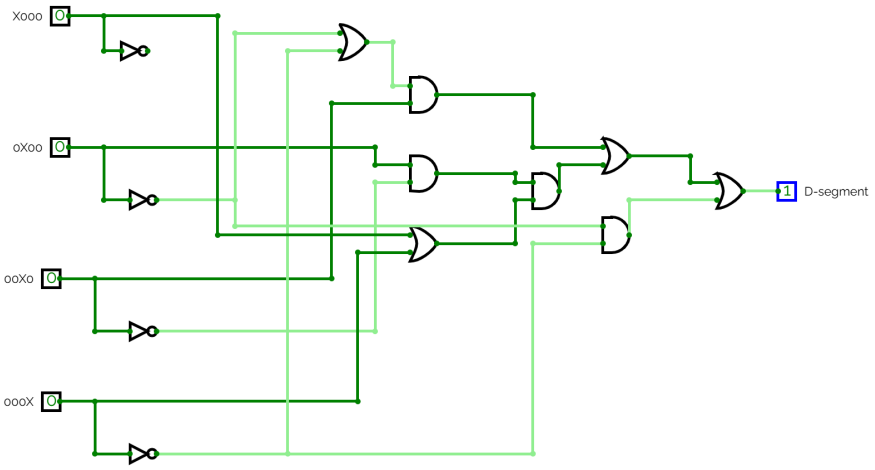


Figure 19: D-segment combinational logic

7.3.6 E-segment combinational logic

Table 6 is a subset of table 1 displaying the individual Boolean state of the E-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	E segment output
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10	1010	1
11	1011	1
12	1100	1
13	1101	1
14	1110	1
15	1111	1

Table 6: E-segment Boolean state based on 4-bit input

Writing table 6 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 20.

D3D2/D1D0	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	1	1	1	1
10	1	0	1	1

$$\overline{D2}\,\overline{D0} + D3D2 + D3D1 + D1\overline{D0}$$

$$D3D2 + D3D1 + \overline{D2}\,\overline{D0} + D1\overline{D0}$$

$$D3(D2 + D1) + \overline{D0}(\overline{D2} + D1)$$

Figure 20: E-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 21.

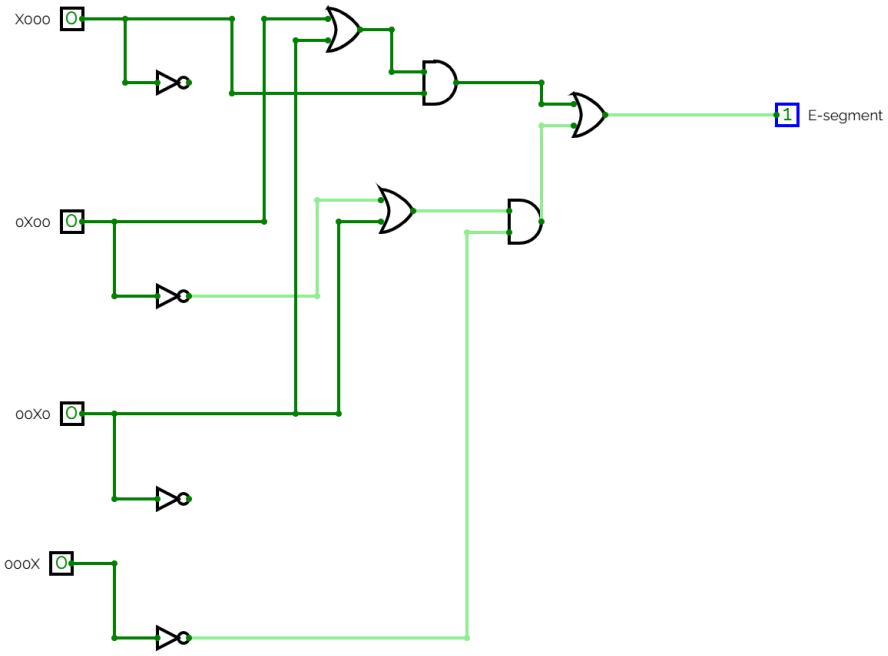


Figure 21: E-segment combinational logic

7.3.7 F-segment combinational logic

Table 7 is a subset of table 1 displaying the individual Boolean state of the F-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	F segment output
0	0000	1
1	0001	0
2	0010	0
3	0011	0
4	0100	1
5	0101	1
6	0110	1
7	0111	0
8	1000	1
9	1001	1
10	1010	0
11	1011	1
12	1100	1
13	1101	0
14	1110	1
15	1111	1

Table 7: F-segment Boolean state based on 4-bit input

Writing table 7 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 22.

D3D2/D1D0	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	1	0	1	1
10	1	1	1	0

$$\overline{D1}\,\overline{D0} + D2\overline{D0} + \overline{D3}D2\overline{D1} + D3\overline{D2}D0 + D3D1D0$$

$$\overline{D0}(\overline{D1} + D2) + \overline{D3}D2\overline{D1} + D3D0(\overline{D2} + D1)$$

Figure 22: F-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 23.

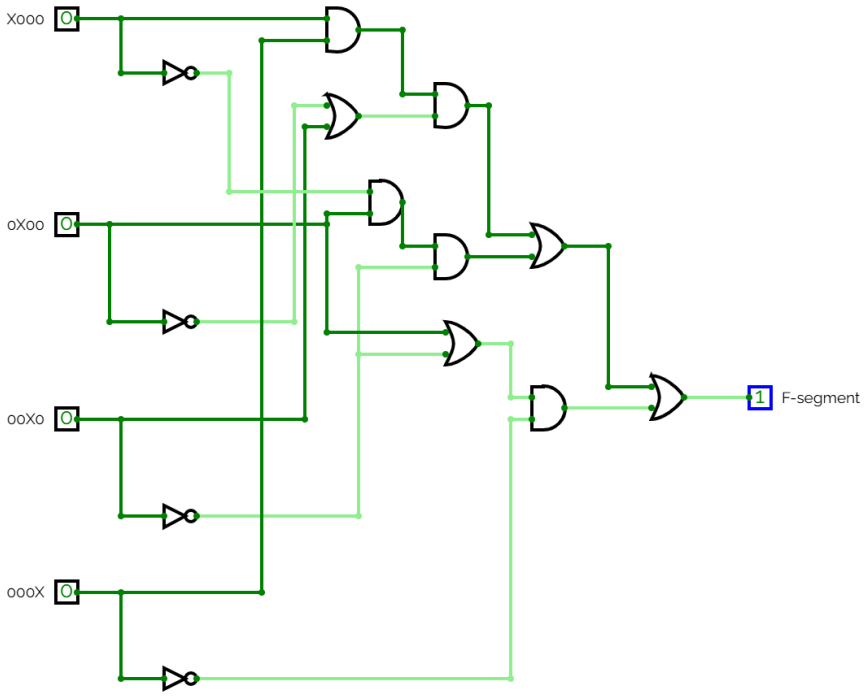


Figure 23: F-segment combinational logic

7.3.8 G-segment combinational logic

Table 8 is a subset of table 1 displaying the individual Boolean state of the G-segment based on the binary input from the motherboard.

Decimal representation	Input (D3D2D1D0)	G segment output
0	0000	0
1	0001	0
2	0010	1
3	0011	1
4	0100	1
5	0101	1
6	0110	1
7	0111	0
8	1000	1
9	1001	1
10	1010	1
11	1011	1
12	1100	0
13	1101	1
14	1110	1
15	1111	1

Table 8: B-segment Boolean state based on 4-bit input

Writing table 8 in a 4x4 Karnaugh map using gray code produces the following combinational logic shown in figure 24.

D3D2/D1D0	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	0	1	1	1
10	1	1	1	1

$$\overline{D3}D2\overline{D1} + D3\overline{D2} + D3D0 + \overline{D2}D1 + D1\overline{D0}$$

$$D3D0 + D3\overline{D2} + \overline{D2}D1 + D1\overline{D0} + \overline{D3}D2\overline{D1}$$

$$D3(D0 + \overline{D2}) + D1(\overline{D2} + \overline{D0}) + \overline{D3}D2\overline{D1}$$

Figure 24: G-segment Karnaugh map + digital logic simplification

Implementing the logic above using logic gates yields the following circuit shown in figure 25.

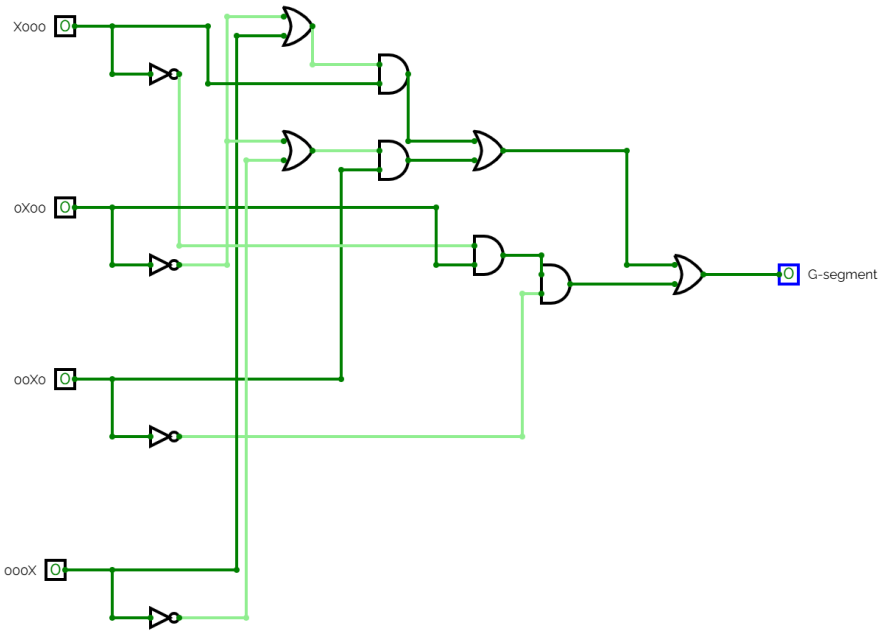


Figure 25: G-segment combinational logic

7.3.9 Combinational logic simulation

Before designing the physical PCB to decode binary into 7SD control signals, I ran my derived combinational logic through a simulator to ensure the base logic was sound. Check out my simulation on CircuitVerse here.

7.4 Motherboard Design

The motherboard design is integral to the functionality of the entire project. The motherboard is what allows all control signals and data signals to interface with the EEPROM hardware. The motherboard also hosts a suite of LEDs that show the status of each address, data, and control signal trace.

The motherboard docks with all daughterboards, taking signals from the MCU and directing them to the shift register, then taking the shift register signals and debug board signals through a multiplexer into the seven-segment decoder and ZIF (zero input force) socket which houses the EEPROM to be programmed.

The EEPROM programmer motherboard can be represented using the following functional block diagram in figure 26:

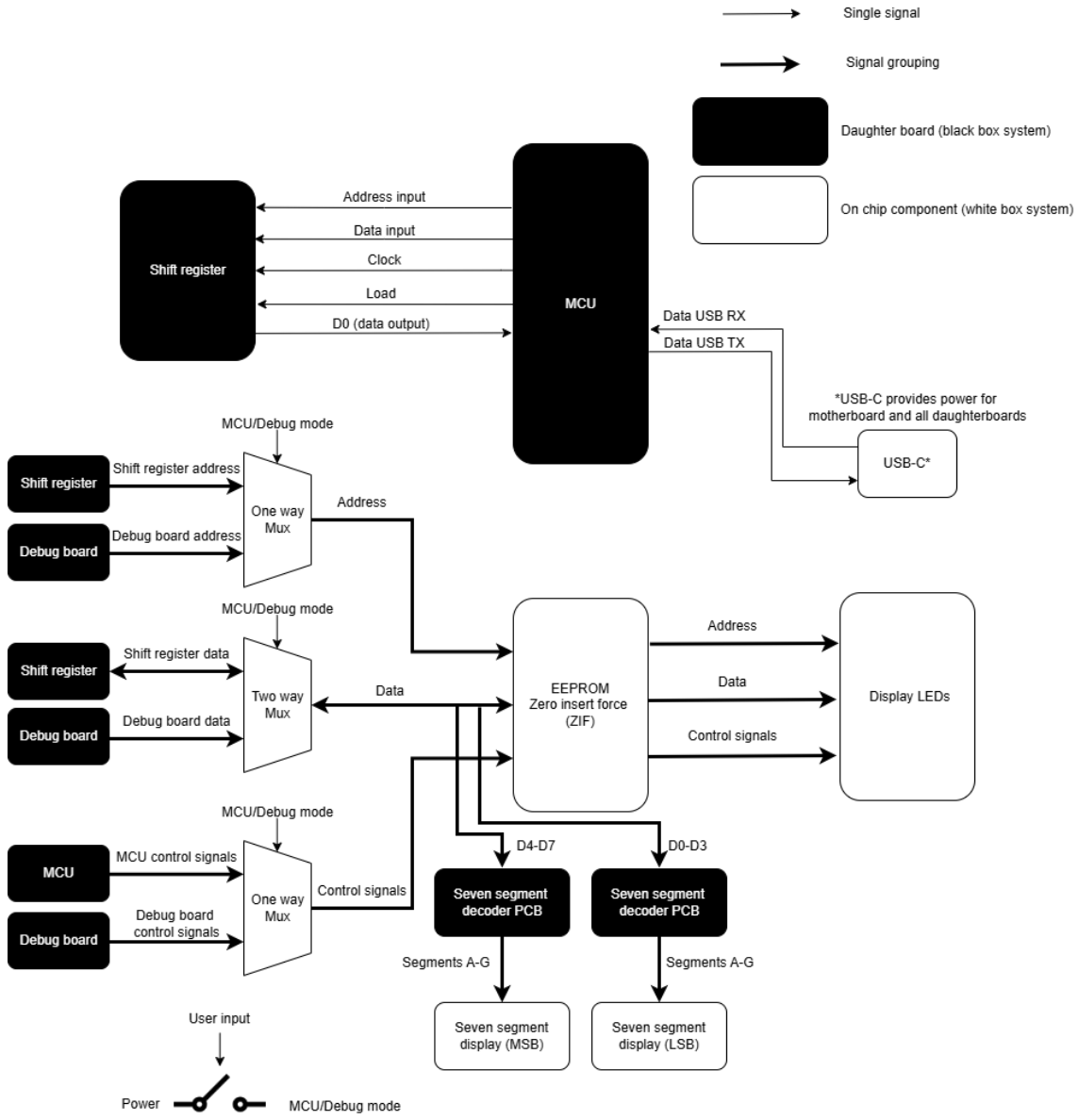


Figure 26: EEPROM programmer motherboard functional block diagram

As seen in the functional block diagram, the MCU/Debug mode switch allows the EEPROM Programmer to be controlled through either the MCU (Arduino) or the manual debug board satisfying R1 (from section 5.1). The motherboard also has a LED display showing the status of all address, data, and control signals in addition with a seven-segment display to humanize the data read out satisfying R2 (from section 5.2). While not shown on any block diagram, all PCBs have a power LED indicator which illuminates when the board is powered satisfying R3 (from section 5.3). Through the development of the shift register, the MCU can serially read and write data allowing for the minimization of required GPIOs thus keeping the list of compatible MCUs larger than if the MCU had to interface in parallel with the EEPROM satisfying R4 (from section 5.4).

Details on how R5 (from section 5.5) and R6 (from section 5.6) are satisfied are discussed in later sections.

Unlike the previous daughterboards, the motherboard doesn't necessarily "do" anything, rather it connects everything together and interfaces with the user through the "MCU/Debug mode" switch to

alternate between MCU control and human manual control.

8 Bill of Materials Formation and Considerations

The project component cost according to the BOM ended up coming out to $196.70 CAD however I already had ZIF (zero input force) sockets and EEPROMs from a previous order so I was able to cut the raw cost down to $183.69 CAD. I sourced all parts for this project using Digikey. Digikey has a price break feature where if you buy a greater quantity of items, the cost per unit decreases. In effect, buying ten of one item is usually cheaper than buying nine. By strategically leveraging this price break feature on a case by case basis, I was able to bring down my raw cost from $183.69 CAD down to $119.32 CAD, a 35% reduction in component costs.

Below is a table of all components required to build the EEPROM programmer before changing the quantities to take advantage of price breaks.

TABLE INCOMPLETE, coming soon.