Journal of Research High School

Vol. 2024 (12) 550 – 559
ISSN 2688-3651 [online]

An ASIC Implementation of ASCII-to-Braille Conversion for Electrical/Mechanical Braille Reading Applications

Riley Ruilin Gu[1] *

[1]Lynbrook High School, San Jose, CA, USA
*Corresponding Author: rileyisgu@gmail.com

Advisor: Michael McGivern, mmcgivern@mitre.org

Received August 21, 2024; Revised September 21, 2024; Accepted October 5, 2024

Abstract

A single-chip Application-Specific Integrated Circuit (ASIC) for text-to-Braille conversion has been proposed to address the limitations of existing solutions, such as limited translation abilities, high costs, and low reliability. This chip is designed to support both electrical (via light-emitting diodes) and mechanical (using push-pull solenoid actuators) Braille character displays. The ASIC integrates various components, including a memory block, a character size calculator, a mapping table, a converter, and a reader, all developed using the Verilog hardware description language (HDL). It is capable of simultaneously displaying up to 8 Braille characters and introduces a novel feature—reading pace control—to enhance usability. The text-to-Braille conversion function has been successfully simulated in Verilog and verified through Field-Programmable Gate Array (FPGA) implementation. The chip is fabricated using the SkyWater 130nm Complementary Metal-Oxide-Semiconductor (CMOS) process, with 5 metal layers, through the open-source Tiny Tapeout program. This solution provides a fully-translatable, cost-effective, and mass-manufacturable design, serving as a scalable logic processor base that can be expanded to display additional Braille characters concurrently.

Keywords: ASIC, ASCII-to-Braille conversion Chip, Integrated Circuit, FPGA, Verilog

1. Introduction

The emergence of artificial intelligence (AI) has triggered an explosive increase of data exchange within both the physical and virtual worlds. Society has greatly benefited from the significant changes brought by AI. However, current data exchange primarily relies on texts, images, or videos, which limits accessibility for people with impaired vision.

To assist individuals with impaired vision in reading text in real time, several hardware-based text-to-Braille conversion systems have been proposed. Letters, digits, and punctuation marks in the digital world are represented by the American Standard Code for Information Interchange (ASCII), therefore text-to-Braille conversion in a digital medium is often referred to as ASCII-to-Braille conversion. Zhang et al. (2006, 2007) introduced a Field-Programmable Gate Array (FPGA) solution capable of translating ASCII text into contracted Braille (Blenkhorn, 1997; Slaby, 1990). This design, while innovative, simplifies implementation by outputting Braille contractions rather than character-by-character outputs. While this technology proves quite useful, it is hindered by high FPGA costs, not being a practical option for those learning Braille, and requiring updates for new Braille contractions.

In contrast, Kumari et al. (2020) proposed a solution using a single-board computer (Raspberry Pi), which directly translates text-images into Braille code. This approach relies on an optical character recognition (OCR) system to detect characters from the text and then sending the character images to the Raspberry Pi for Braille translation. Despite its functionality, this is primarily a software-based system, and the reliance on OCR limits its use in digital information exchange. Furthermore, incorporating a computer like the Raspberry Pi into the image-to-Braille conversion process is neither cost-effective nor energy efficient.

Saxena et al. (2022) proposed a hybrid solution combining a discrete-circuit with a Raspberry Pi. In this model, the ASCII-to-Braille conversion table is implemented using discrete circuitry, while other controls are managed through software. However, this design supports only uppercase-to-lowercase character translation due to circuit size limitations. Moreover, the complexity of the discrete circuit raises concerns about long-term reliability.

The common drawback of all these solutions is the lack of user control over reading speed, which is crucial to the reading experience for individuals with impaired vision.

To address the above issues in the existing solutions, this work focuses on proposing a single chip solution, Application-Specific Integrated Circuit (ASIC), to implement low-cost, high-speed, character-by-character ASCII-to-Braille conversion that consumes less power and has higher reliability. A button for reading pace control is implemented, allowing readers to go through sets of characters at their desired pace. For different purposes, this ASIC can drive either light-emitting diodes (LED) for prototyping or push-pull solenoid actuators (PPSA) for final deployment.

This paper is organized as follows: Section 2 introduces the Braille alphabet and its binary representation. Section 3 describes the design details of proposed ASIC for ASCII-to-Braille conversion, including simulation results. FPGA experimental results are presented in Section 4. Finally, a discussion and conclusion are provided in Section 5 and 6, respectively.

2. Background of the Braille Alphabet

The Braille alphabet was invented by Louis Braille in the early 19th century as a tactile reading and writing system for those who are visually impaired, blind, or deaf-blind. The language uses raised dots and flat dots arranged in cells, with each cell consisting of six dots organized in a rectangular grid of two columns and three rows. This system allows for unique six-dot configurations that represent letters, digits, punctuation marks, and various indicator characters that are required for clarity. Fig 1. illustrates a basic scope of how common characters are described through Braille and will serve as the basis for translation.

In this work, each dot will be replaced with either LED or PPSA. For example, when LED light is on, it represents a raised dot. When the LED light is off, it represents a flat dot. The green, red, or white LED is chosen for the best visual recognition (Fig.2).

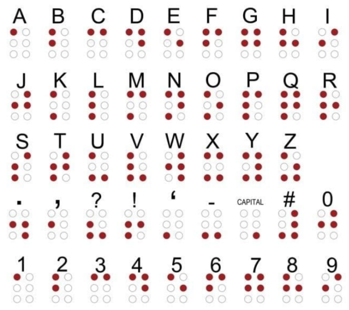


Figure 1. Braille alphabet

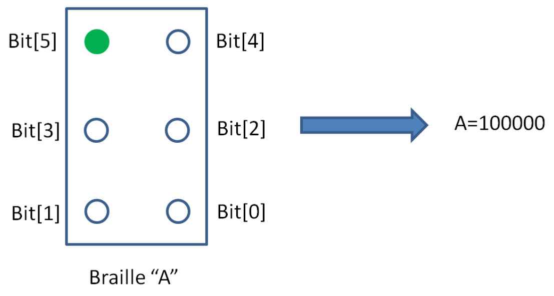


Figure 2. Binary-coding for Braille character

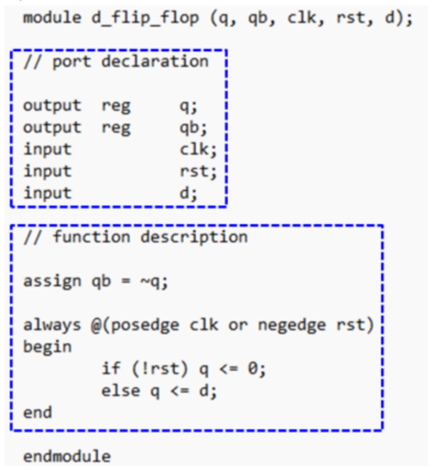
In order to display Braille characters with LEDs or PPSAs, a binary code for each Braille character is developed. Following the rule that the raised dot is represented by "1" and the flat dot is represented by "0", Fig.2 demonstrates the braille letter "A" encoded into a 6-bit binary number, bit[5:0]=100000. Beginning from the upper left corner, each dot is assigned to its binary bit left-to-right, row-by-row. The other characters are encoded into binary the same way. However, capital and lower-case letters as well as a few digits and letters have the same braille configuration (e.g. '3' & 'c'). To resolve this conflict, two indicators are used to distinguish capital letters and digits from their matching counterparts. These two indicators, CAPITAL → 000001b and DIGIT → 010111b, must come before its corresponding letter or digit to indicate what the following character will be. Therefore, capital letters and digits require two Braille cells for accurate Braille readings.

3. ASIC Implementation of ASCII-to-Braille Conversion

3.1 Methodology: Hardware Description Language (HDL) and ASIC design flow

Hardware description language, Verilog-HDL, is used to implement this ASCII-to-Braille ASIC (Nelson et al., 1995). It describes the data flow and timing of a circuit at high level, register-transfer-level (RTL), without being tied to the fabrication process (Palnitkar, 1996). In RTL, each building block in ASIC is designed/implemented as a "module" which includes the input/output pin declaration and function description (Fig.3). A top "module" will contain all building blocks and their interconnections to get the whole ASIC RTL.

As illustrated in Fig.4, after the function of RTL design is verified with Verilog simulator, RTL will be synthesized to gate-level design, a low level design including logic gates (inverter, nand, nor, flip-flop,…) , to tie to a selected fabrication process. Following the design rule of this selected fabrication process, gate-level design will be further converted to physical layout though place-and-route tool. Finally, the ASIC will be fabricated based on this physical layout.



```
module d_flip_flop (q, qb, clk, rst, d);

// port declaration

output  reg     q;
output  reg     qb;
input           clk;
input           rst;
input           d;

// function description

assign qb = ~q;

always @(posedge clk or negedge rst)
begin
        if (!rst) q <= 0;
        else q <= d;
end

endmodule
```

Figure 3. Example of Verilog module

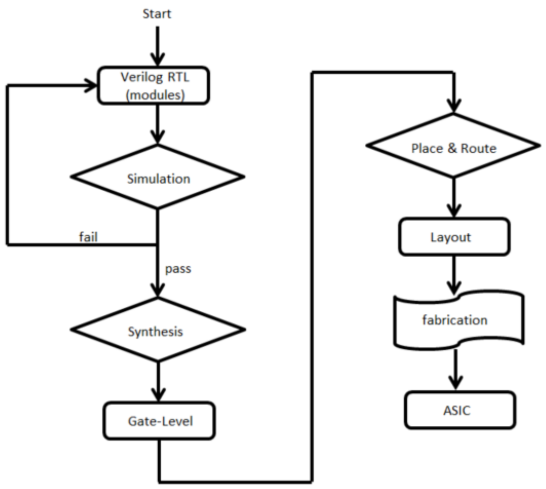


Figure.4 ASIC design flow

3.2 Architecture and operation of ASCII-to-Braille conversion

The block diagram of the proposed ASIC is shown in Fig. 5. The ASIC contains five building blocks: memory, ASCII-to-Braille mapping table, character size calculator, ASCII-to-Braille converter, Braille buffer and reader. All of these components are implemented using Verilog-HDL and simulated using Icarus Verilog (IIC-JKU, 2022), an open-source Verilog simulator.

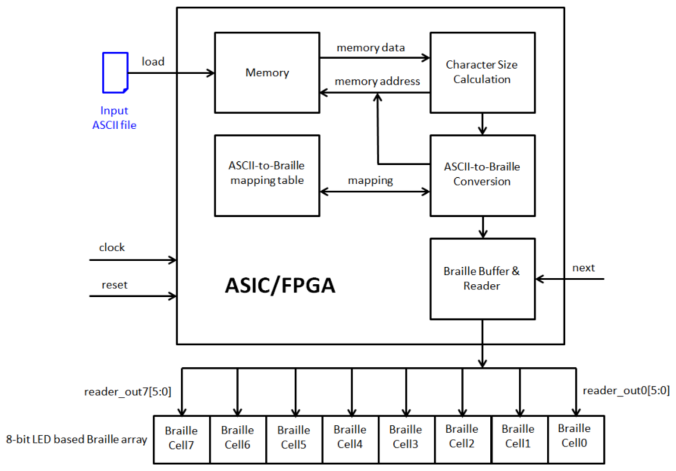


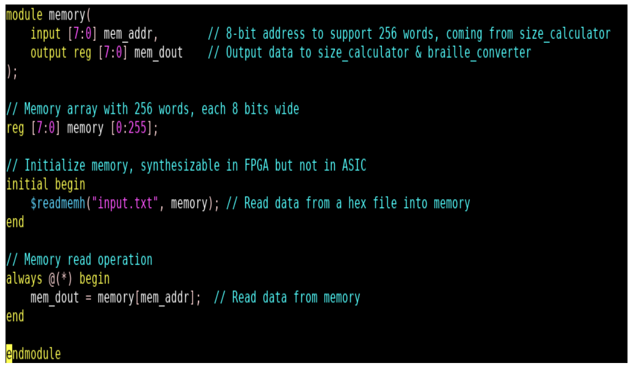
Figure 5. Block diagram of proposed ASIC

The input text file to be translated into Braille is first loaded into memory in ASCII format. The character size calculator then scans the memory to determine the number of ASCII characters, as well as the number of capital letters and digits present. As a result of this process, two character sizes are calculated: the ASCII character size (the number of ASCII characters) and the Braille character size (the ASCII character size plus the number of indicator characters). Once the character size calculation is completed, the ASCII-to-Braille converter begins its operation. The converter scans the memory as well, translating each ASCII character into its corresponding Braille character, adding an indicator character for each capital letter and digit, and saving every Braille character into a buffer until all ASCII characters have been processed. By sending a "next" pulse to the reader, 8 Braille characters are transferred from the buffer to an LED-based Braille cell array for display. When the user sends another "next" pulse, the next 8 Braille characters are sent to the LED-based Braille cell array, and so on, allowing the user to control the reading pace.

3.3 Design details of building blocks

Memory

Fig. 6 illustrates the design of the memory. This memory can store up to 256 bytes (ASCII characters), with each byte consisting of 8 bits. It is initialized by loading the input ASCII file. Once the memory receives an address signal from either the size calculator or the ASCII-to-Braille converter, the data stored at that address will be sent to the size calculator and the ASCII-to-Braille converter.

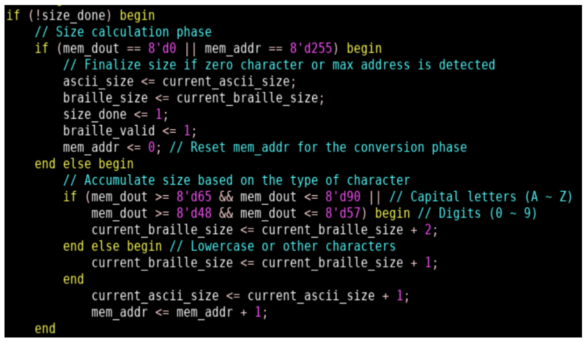


```verilog
module memory(
    input [7:0] mem_addr,       // 8-bit address to support 256 words, coming from size_calculator
    output reg [7:0] mem_dout   // Output data to size_calculator & braille_converter
);

// Memory array with 256 words, each 8 bits wide
reg [7:0] memory [0:255];

// Initialize memory, synthesizable in FPGA but not in ASIC
initial begin
    $readmemh("input.txt", memory); // Read data from a hex file into memory
end

// Memory read operation
always @(*) begin
    mem_dout = memory[mem_addr];  // Read data from memory
end

endmodule
```

Figure 6. Memory implementation

Character size calculator

Fig. 7 illustrates the design of the character size calculator. When the "reset" signal is asserted (0), the memory address, ASCII size, Braille size, and size_done flag are all cleared to 0. When the "reset" signal is released (1), the size calculator first checks if the size_done flag is set (1). If the size_done flag is not set (0), the size calculator begins updating the memory address and checks the memory data at the updated address during each clock cycle. If the memory data is between 65d and 90d, it represents a capital letter (‘A’ to ‘Z’). If the memory data is between 48d and 57d, it represents a digit (‘0’ to ‘9’). In these two cases, the Braille size is incremented by 2 to account for the indicator characters. For lowercase letters and other characters, the Braille size increments by 1 every clock cycle, while the ASCII size consistently increases by 1 during each clock cycle, regardless of the character type.



```verilog
if (!size_done) begin
    // Size calculation phase
    if (mem_dout == 8'd0 || mem_addr == 8'd255) begin
        // Finalize size if zero character or max address is detected
        ascii_size <= current_ascii_size;
        braille_size <= current_braille_size;
        size_done <= 1;
        braille_valid <= 1;
        mem_addr <= 0; // Reset mem_addr for the conversion phase
    end else begin
        // Accumulate size based on the type of character
        if (mem_dout >= 8'd65 && mem_dout <= 8'd90 || // Capital letters (A ~ Z)
            mem_dout >= 8'd48 && mem_dout <= 8'd57) begin // Digits (0 ~ 9)
            current_braille_size <= current_braille_size + 2;
        end else begin // Lowercase or other characters
            current_braille_size <= current_braille_size + 1;
        end
            current_ascii_size <= current_ascii_size + 1;
            mem_addr <= mem_addr + 1;
    end
```

Figure 7. Implementation of the character size calculator

When the memory data is 0 or the memory address reaches 255d, it indicates the end of the input ASCII file. At this point, the two character sizes stop increasing and are saved into the ascii_size and braille_size registers, respectively. The size_done flag is then set to 1, and the memory address is reset to 0.

In the simulation waveforms shown in Fig. 8, there are 14 ASCII characters (43h, 61h, 74h, 54h, 6Fh, 6Dh, 6Ch, 69h, 6Fh, 6Eh, 6Bh, 69h, 6Eh, 67h) stored in memory addresses 00h~0Dh (0d~13d). Therefore, the ascii_size is 0Eh (14d). However, since the ASCII characters 43h and 54h represent the capital letters "C" and "T," respectively, the braille_size becomes 10h (16d) to account for the inclusion of two capital indicators.

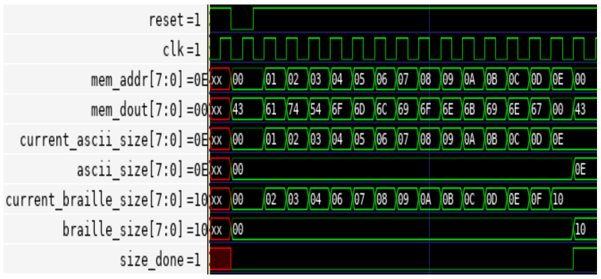
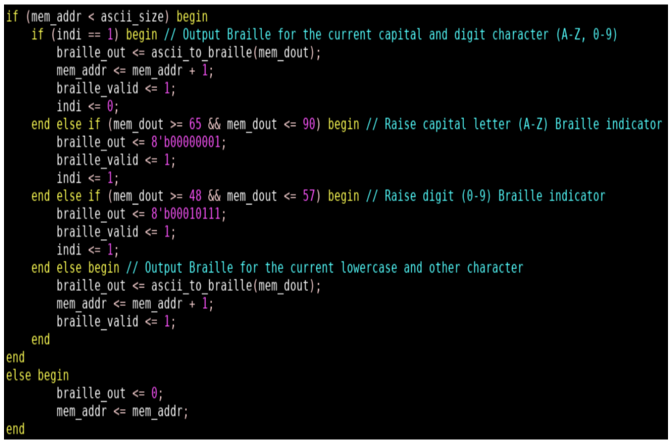


Figure 8. Simulation waveforms of the character size calculator

Vol. 2024 (12) 550 – 559
ISSN 2688-3651 [online]

ASCII-to-Braille converter and ASCII-to-Braille mapping table

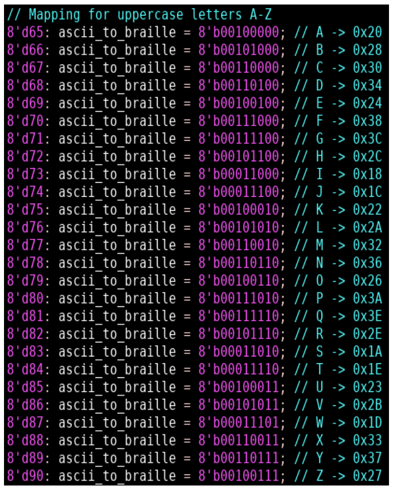
Fig. 9 illustrates the design of the ASCII-to-Braille converter. Since the memory address is reset to 0 when the size calculation is completed, the converter begins translating the ASCII data to Braille code starting from the first ASCII character in memory. When the memory data falls within the ranges 65d-90d (for capital letters) or 48d-57d (for digits), the converter first outputs a capital Braille indicator (00000001b) or a digit Braille indicator (00010111b), along with an indicator flag, indi, while keeping the memory address unchanged. In the next clock cycle, based on the indi flag, the ASCII data at the same memory address is mapped to its corresponding Braille code using the mapping tables shown in Fig. 10 (a, c). Thus, the conversion of capital letters and digits requires two clock cycles, first to provide the indicator character and the second for the character code itself. The conversion of lowercase letters and punctuation marks does not require a Braille indicator in front of them and is directly mapped to Braille code in one clock cycle using the mapping tables shown in Fig. 10 (b, c).



```
if (mem_addr < ascii_size) begin
    if (indi == 1) begin // Output Braille for the current capital and digit character (A-Z, 0-9)
        braille_out <= ascii_to_braille(mem_dout);
        mem_addr <= mem_addr + 1;
        braille_valid <= 1;
        indi <= 0;
    end else if (mem_dout >= 65 && mem_dout <= 90) begin // Raise capital letter (A-Z) Braille indicator
        braille_out <= 8'b00000001;
        braille_valid <= 1;
        indi <= 1;
    end else if (mem_dout >= 48 && mem_dout <= 57) begin // Raise digit (0-9) Braille indicator
        braille_out <= 8'b00010111;
        braille_valid <= 1;
        indi <= 1;
    end else begin // Output Braille for the current lowercase and other character
        braille_out <= ascii_to_braille(mem_dout);
        mem_addr <= mem_addr + 1;
        braille_valid <= 1;
    end
end
else begin
        braille_out <= 0;
        mem_addr <= mem_addr;
end
```

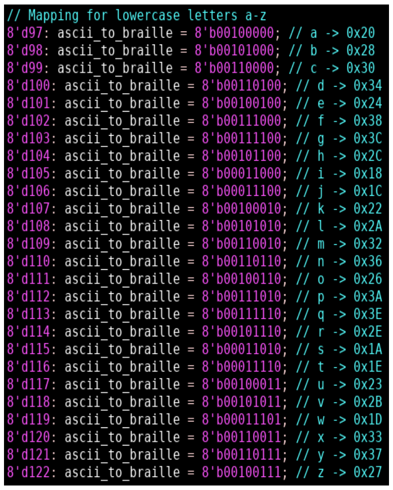
Figure 9. Implementation of ASCII-to-Braille conversion

The braille_valid signal is used to inform the reader when the converter output is valid. Before the memory address reaches ascii_size (which is determined by the size calculator), braille_valid remains at 1. Once the conversion is completed (when the memory address reaches ascii_size), braille_valid returns to 0, the converter continuously outputs 0, and the memory address stays at ascii_size.



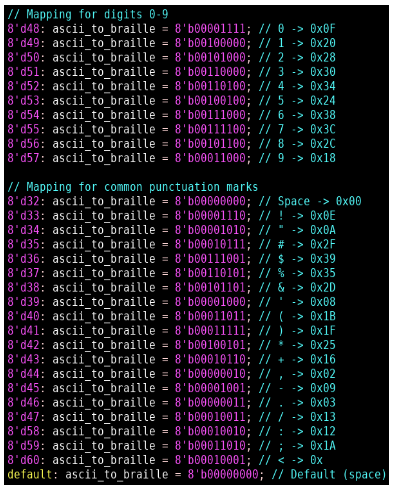
```
// Mapping for uppercase letters A-Z
8'd65: ascii_to_braille = 8'b00100000; // A -> 0x20
8'd66: ascii_to_braille = 8'b00101000; // B -> 0x28
8'd67: ascii_to_braille = 8'b00110000; // C -> 0x30
8'd68: ascii_to_braille = 8'b00110100; // D -> 0x34
8'd69: ascii_to_braille = 8'b00100100; // E -> 0x24
8'd70: ascii_to_braille = 8'b00111000; // F -> 0x38
8'd71: ascii_to_braille = 8'b00111100; // G -> 0x3C
8'd72: ascii_to_braille = 8'b00101100; // H -> 0x2C
8'd73: ascii_to_braille = 8'b00011000; // I -> 0x18
8'd74: ascii_to_braille = 8'b00011100; // J -> 0x1C
8'd75: ascii_to_braille = 8'b00100010; // K -> 0x22
8'd76: ascii_to_braille = 8'b00101010; // L -> 0x2A
8'd77: ascii_to_braille = 8'b00110010; // M -> 0x32
8'd78: ascii_to_braille = 8'b00110110; // N -> 0x36
8'd79: ascii_to_braille = 8'b00100110; // O -> 0x26
8'd80: ascii_to_braille = 8'b00111010; // P -> 0x3A
8'd81: ascii_to_braille = 8'b00111110; // Q -> 0x3E
8'd82: ascii_to_braille = 8'b00101110; // R -> 0x2E
8'd83: ascii_to_braille = 8'b00011010; // S -> 0x1A
8'd84: ascii_to_braille = 8'b00011110; // T -> 0x1E
8'd85: ascii_to_braille = 8'b00100011; // U -> 0x23
8'd86: ascii_to_braille = 8'b00101011; // V -> 0x2B
8'd87: ascii_to_braille = 8'b00011101; // W -> 0x1D
8'd88: ascii_to_braille = 8'b00110011; // X -> 0x33
8'd89: ascii_to_braille = 8'b00110111; // Y -> 0x37
8'd90: ascii_to_braille = 8'b00100111; // Z -> 0x27
```

Figure 10(a). ASCII-to-Braille mapping table, uppercase letters



```
// Mapping for lowercase letters a-z
8'd97: ascii_to_braille = 8'b00100000; // a -> 0x20
8'd98: ascii_to_braille = 8'b00101000; // b -> 0x28
8'd99: ascii_to_braille = 8'b00110000; // c -> 0x30
8'd100: ascii_to_braille = 8'b00110100; // d -> 0x34
8'd101: ascii_to_braille = 8'b00100100; // e -> 0x24
8'd102: ascii_to_braille = 8'b00111000; // f -> 0x38
8'd103: ascii_to_braille = 8'b00111100; // g -> 0x3C
8'd104: ascii_to_braille = 8'b00101100; // h -> 0x2C
8'd105: ascii_to_braille = 8'b00011000; // i -> 0x18
8'd106: ascii_to_braille = 8'b00011100; // j -> 0x1C
8'd107: ascii_to_braille = 8'b00100010; // k -> 0x22
8'd108: ascii_to_braille = 8'b00101010; // l -> 0x2A
8'd109: ascii_to_braille = 8'b00110010; // m -> 0x32
8'd110: ascii_to_braille = 8'b00110110; // n -> 0x36
8'd111: ascii_to_braille = 8'b00100110; // o -> 0x26
8'd112: ascii_to_braille = 8'b00111010; // p -> 0x3A
8'd113: ascii_to_braille = 8'b00111110; // q -> 0x3E
8'd114: ascii_to_braille = 8'b00101110; // r -> 0x2E
8'd115: ascii_to_braille = 8'b00011010; // s -> 0x1A
8'd116: ascii_to_braille = 8'b00011110; // t -> 0x1E
8'd117: ascii_to_braille = 8'b00100011; // u -> 0x23
8'd118: ascii_to_braille = 8'b00101011; // v -> 0x2B
8'd119: ascii_to_braille = 8'b00011101; // w -> 0x1D
8'd120: ascii_to_braille = 8'b00110011; // x -> 0x33
8'd121: ascii_to_braille = 8'b00110111; // y -> 0x37
8'd122: ascii_to_braille = 8'b00100111; // z -> 0x27
```

Figure 10(b). ASCII-to-Braille mapping table, lowercase letters



```
// Mapping for digits 0-9
8'd48: ascii_to_braille = 8'b00001111; // 0 -> 0x0F
8'd49: ascii_to_braille = 8'b00100000; // 1 -> 0x20
8'd50: ascii_to_braille = 8'b00101000; // 2 -> 0x28
8'd51: ascii_to_braille = 8'b00110000; // 3 -> 0x30
8'd52: ascii_to_braille = 8'b00110100; // 4 -> 0x34
8'd53: ascii_to_braille = 8'b00100100; // 5 -> 0x24
8'd54: ascii_to_braille = 8'b00111000; // 6 -> 0x38
8'd55: ascii_to_braille = 8'b00111100; // 7 -> 0x3C
8'd56: ascii_to_braille = 8'b00101100; // 8 -> 0x2C
8'd57: ascii_to_braille = 8'b00011000; // 9 -> 0x18

// Mapping for common punctuation marks
8'd32: ascii_to_braille = 8'b00000000; // Space -> 0x00
8'd33: ascii_to_braille = 8'b00001110; // ! -> 0x0E
8'd34: ascii_to_braille = 8'b00001010; // " -> 0x0A
8'd35: ascii_to_braille = 8'b00010111; // # -> 0x2F
8'd36: ascii_to_braille = 8'b00111001; // $ -> 0x39
8'd37: ascii_to_braille = 8'b00110101; // % -> 0x35
8'd38: ascii_to_braille = 8'b00101101; // & -> 0x2D
8'd39: ascii_to_braille = 8'b00001000; // ' -> 0x08
8'd40: ascii_to_braille = 8'b00011011; // ( -> 0x1B
8'd41: ascii_to_braille = 8'b00011111; // ) -> 0x1F
8'd42: ascii_to_braille = 8'b00100101; // * -> 0x25
8'd43: ascii_to_braille = 8'b00010110; // + -> 0x16
8'd44: ascii_to_braille = 8'b00000010; // , -> 0x02
8'd45: ascii_to_braille = 8'b00001001; // - -> 0x09
8'd46: ascii_to_braille = 8'b00000011; // . -> 0x03
8'd47: ascii_to_braille = 8'b00010011; // / -> 0x13
8'd58: ascii_to_braille = 8'b00010010; // : -> 0x12
8'd59: ascii_to_braille = 8'b00011010; // ; -> 0x1A
8'd60: ascii_to_braille = 8'b00010001; // < -> 0x
default: ascii_to_braille = 8'b00000000; // Default (space)
```

Figure 10(c). ASCII-to-Braille mapping table, digits & punctuation marks

In the simulation waveforms shown in Fig. 11, the memory ASCII data 43h (67d in the mapping table, representing the capital letter "C") is converted into two Braille codes: the capital indicator 01h and the letter 30h. Similarly, the memory ASCII data 54h (84d in the mapping table, representing the capital letter "T") is also converted into two Braille codes: the capital indicator 01h and the letter 1Eh. From the memory address/data waveform, you can see that these two conversions require two clock cycles, whereas the other conversions (61h→20h, 74h→1Eh, 6Fh→26h, 6Dh→32h, 6Ch→2Ah, 69h→18h, 6Fh→26h, 6Eh→36h, 6Bh→22h, 69h→18h, 6Eh→36h, 67h→3Ch) each take only one clock cycle. The braille_valid signal remains at 1 to indicate that all Braille outputs are valid.

Braille buffer and reader

The Braille buffer and reader store the received Braille codes in a buffer and send them to the reader to drive the LED or PPSA as requested. This process operates as a 6-state Finite-State Machine (FSM), as illustrated in Fig.12 and Fig.13.

After reset is released, FSM stays in the IDLE state and waits for braille_valid to be flagged. When braille_valid is "1", braille_size is loaded into loaded_braille_size (Fig.12) and the FSM enters the LOADING state (Fig.13).

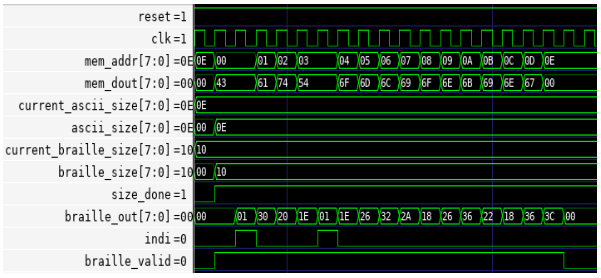
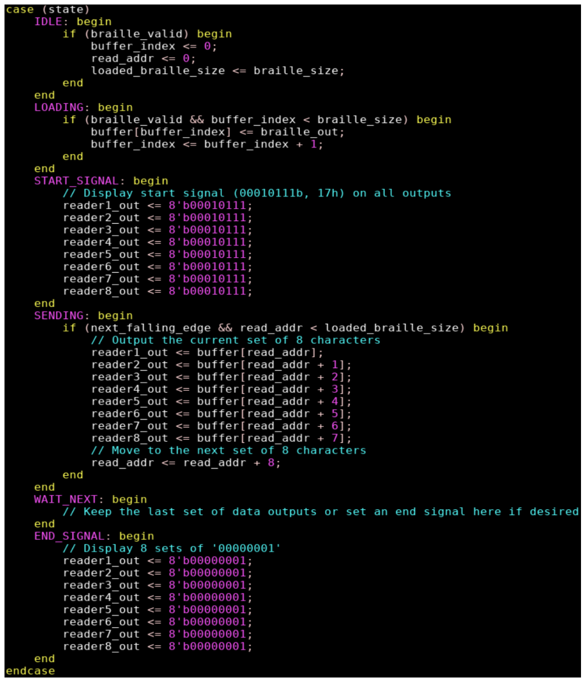
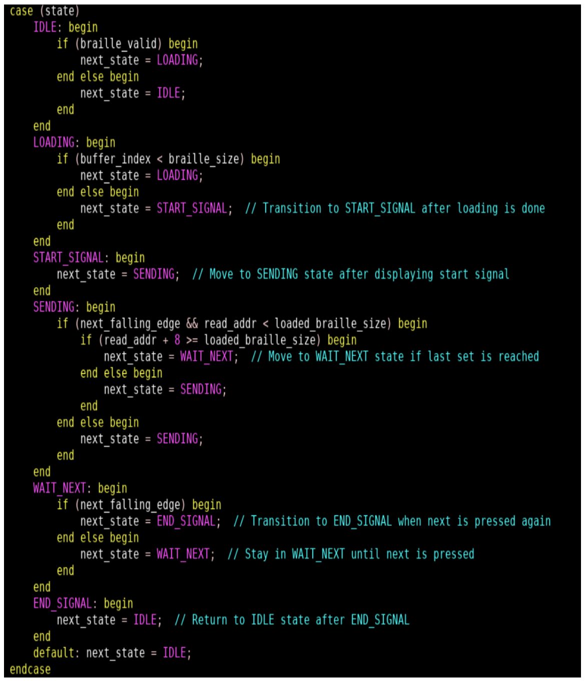


Figure 11. Simulation waveforms of ASCII-to-Braille converter


```
case (state)
    IDLE: begin
        if (braille_valid) begin
            buffer_index <= 0;
            read_addr <= 0;
            loaded_braille_size <= braille_size;
        end
    end
    LOADING: begin
        if (braille_valid && buffer_index < braille_size) begin
            buffer[buffer_index] <= braille_out;
            buffer_index <= buffer_index + 1;
        end
    end
    START_SIGNAL: begin
        // Display start signal (00010111b, 17h) on all outputs
        reader1_out <= 8'b00010111;
        reader2_out <= 8'b00010111;
        reader3_out <= 8'b00010111;
        reader4_out <= 8'b00010111;
        reader5_out <= 8'b00010111;
        reader6_out <= 8'b00010111;
        reader7_out <= 8'b00010111;
        reader8_out <= 8'b00010111;
    end
    SENDING: begin
        if (next_falling_edge && read_addr < loaded_braille_size) begin
            // Output the current set of 8 characters
            reader1_out <= buffer[read_addr];
            reader2_out <= buffer[read_addr + 1];
            reader3_out <= buffer[read_addr + 2];
            reader4_out <= buffer[read_addr + 3];
            reader5_out <= buffer[read_addr + 4];
            reader6_out <= buffer[read_addr + 5];
            reader7_out <= buffer[read_addr + 6];
            reader8_out <= buffer[read_addr + 7];
            // Move to the next set of 8 characters
            read_addr <= read_addr + 8;
        end
    end
    WAIT_NEXT: begin
        // Keep the last set of data outputs or set an end signal here if desired
    end
    END_SIGNAL: begin
        // Display 8 sets of '00000001'
        reader1_out <= 8'b00000001;
        reader2_out <= 8'b00000001;
        reader3_out <= 8'b00000001;
        reader4_out <= 8'b00000001;
        reader5_out <= 8'b00000001;
        reader6_out <= 8'b00000001;
        reader7_out <= 8'b00000001;
        reader8_out <= 8'b00000001;
    end
endcase
```
Figure 12. Functions in each FSM state


```
case (state)
    IDLE: begin
        if (braille_valid) begin
            next_state = LOADING;
        end else begin
            next_state = IDLE;
        end
    end
    LOADING: begin
        if (buffer_index < braille_size) begin
            next_state = LOADING;
        end else begin
            next_state = START_SIGNAL;  // Transition to START_SIGNAL after loading is done
        end
    end
    START_SIGNAL: begin
        next_state = SENDING;  // Move to SENDING state after displaying start signal
    end
    SENDING: begin
        if (next_falling_edge && read_addr < loaded_braille_size) begin
            if (read_addr + 8 >= loaded_braille_size) begin
                next_state = WAIT_NEXT;  // Move to WAIT_NEXT state if last set is reached
            end else begin
                next_state = SENDING;
            end
        end else begin
            next_state = SENDING;
        end
    end
    WAIT_NEXT: begin
        if (next_falling_edge) begin
            next_state = END_SIGNAL;  // Transition to END_SIGNAL when next is pressed again
        end else begin
            next_state = WAIT_NEXT;  // Stay in WAIT_NEXT until next is pressed
        end
    end
    END_SIGNAL: begin
        next_state = IDLE;  // Return to IDLE state after END_SIGNAL
    end
    default: next_state = IDLE;
endcase
```
Figure 13. FSM in Braille buffer and reader

In the LOADING state, as long as buffer_index (address) is less than loaded_braille_size, the Braille codes coming from the converter will be stored into buffer one-by-one at each clock cycle (Fig.12). After the last Braille code is saved, the FSM enters the START_SIGNAL state (Fig.13).

The START_SIGNAL state lasts until a "next" signal sent by the user is detected. During this state, all 8 readers are set to 17h (Fig.12) which indicates the Braille codes are ready for reading. When the "next" pulse is detected, the FSM state is changed to SENDING (Fig.13).

In the SENDING state, when the "next" signal sent by the user is detected, 8 Braille codes stored in the buffer will be read out through 8 readers simultaneously (Fig.12). The user can continue sending "next" signal pulses to read the next 8 Braille codes until the last set of eight Braille codes is read out. Only then will the FSM move to the WAIT_NEXT state (Fig.13).

In WAIT_NEXT state, FSM does nothing but wait for the user to press the "next" button again such that it can tell the user that they have reached the end. When this final "next" signal comes, the FSM enters the END_SIGNAL state (Fig.13) and all 8 readers are cleared to 01h which indicates the end of the reading (Fig.12).

From the simulation waveforms of Fig.14, in LOAD state (1h), the buffer_index keeps increasing at each clock cycle until 0Fh (15d) is reached. This means a total of 16 Braille codes are stored into the buffer. In the START_SIGNAL state (2h), all reader_out signals are set to 17h to indicate the start of reading. In the SENDING

Vol. 2024 (12) 550 – 559
ISSN 2688-3651 [online]

state (3h), when the 1st "next" signal comes, the 1st 8 Braille codes are sent to readers 1~8 in the first read cycle. When the 2nd "NEXT" signal comes, the following 8 Braille codes are sent to readers 1~8 in the second read cycle. Since all Braille codes have been read out, all reader_out signals will be set to 01h once the 3rd "NEXT" signal arrives. This indicates to the user that the reading has finished.

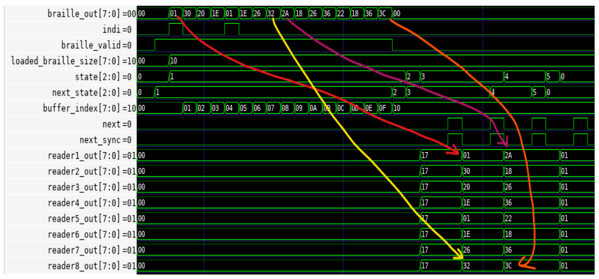


Figure 14. Simulation waveforms of Braille Buffer/Reader

3.4 Synthesis and Place-and-Route (RTL-to-Gate-to-Layout)

The layout database is directly derived from the Verilog RTL using OpenLane (IIC-JKU, 2022), an open-source digital ASIC implementation flow. The process used is the 130nm SkyWater SKY130 CMOS process, which includes 5 metal layers. The layout is shown in Fig.15, and the die area is 401.125 µm × 411.845 µm.

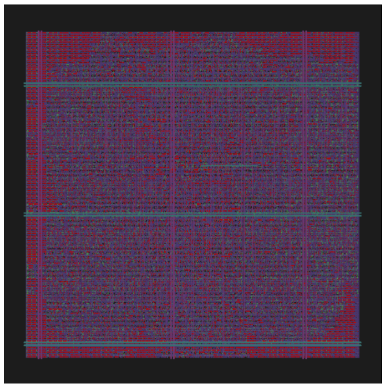


Figure 15. Synthesized layout of the ASCII-to-Braille conversion ASIC

4. Measurement Results

The functionality of this ASIC is verified using an FPGA, as the FPGA design flow closely resembles the ASIC design flow (Fig. 16). In addition to the Verilog RTL developed for the ASIC, a clock generator must be integrated to allow the FPGA to operate independently of an external clock source. Since the logic gates are pre-fabricated within the FPGA, the synthesized gate-level design can be implemented by configuring these existing logic gates through FPGA programming.

As illustrated in Fig. 17(a), the left side shows the ARTY Z7 SoC development board (Digilent, 2020). The Verilog code developed for ASCII-to-Braille conversion is synthesized and programmed into the Xilinx ZYNQ-7000 FPGA (Xilinx, 2018), which is located in the center of the board, to implement the hardware version of the ASCII-to-Braille conversion. Additionally, a 5 MHz clock-wizard IP is instantiated from the Xilinx library and integrated into the ASIC to provide the clock signal for the converter. Due to the limited number of output ports on the ARTY Z7 board, up to 4 Braille cells can be constructed and demonstrated in the experiments. In experiment #1 (Fig. 17a), 6 LEDs are used to construct one Braille cell. In experiment #2 (Fig. 17b), 24 LEDs are used to construct 4 Braille cells. (The LEDs can be replaced with PPSAs for mechanical reading; however, in this measurement, only LEDs are used.)

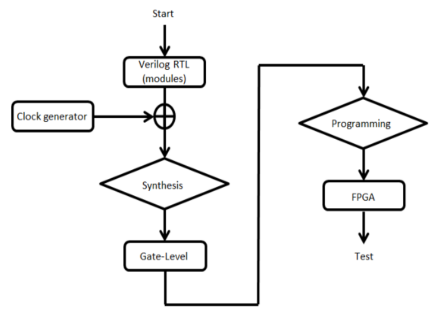


Figure 16. FPGA design flow

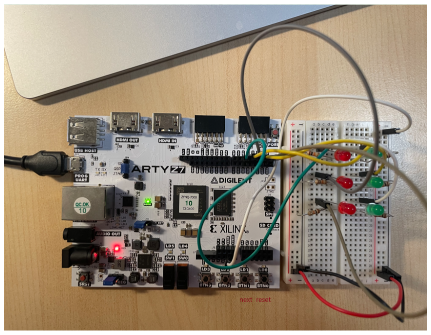


Figure 17(a). Single Braille cell FPGA setup

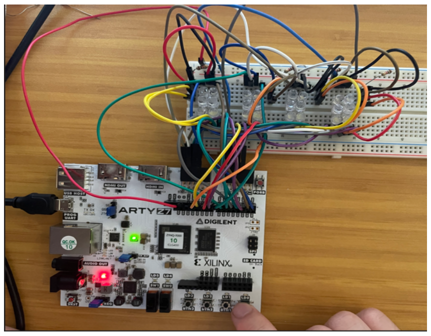


Figure 17(b). Quad Braille cell FPGA setup

In experiment #1, the text *"Hello, bwsi!"* is programmed into memory during FPGA synthesis and programming. This simple experiment utilized only one display cell, therefore displaying one character at a time from the braille code buffer. A single LED Braille cell will display this text character by character. Through the ASCII-to-Braille conversion performed by the ASIC, the translated Braille codes are shown in Fig. 18.

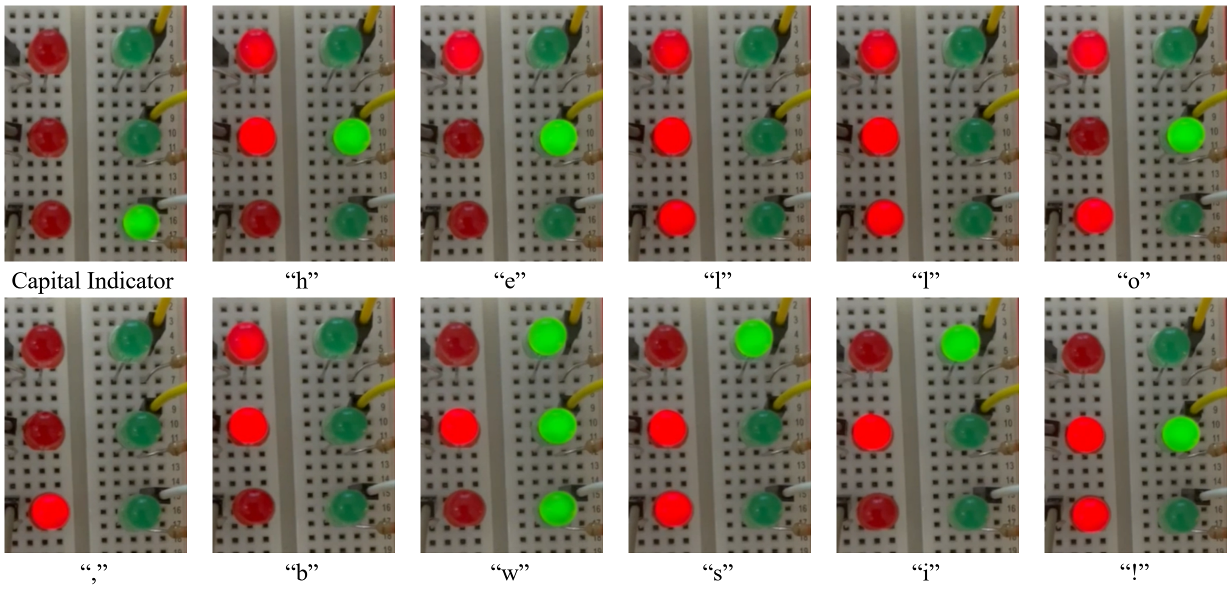
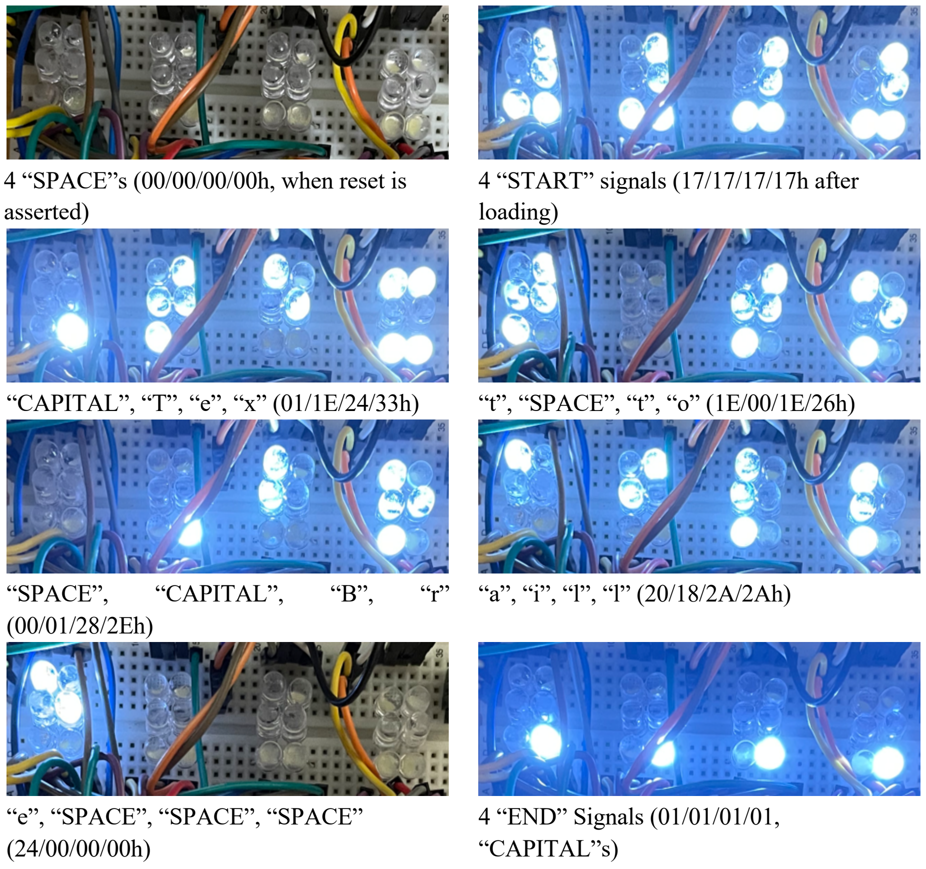


Figure 18. One LED Braille cell display of “Hello, bwsi!”

In experiment #2, text “*Text to Braille*” is programmed into memory after initialization. Four LED Braille cells can display 4 Braille characters at one time. When reset is asserted, all 4 cells display “SPACE”. After reset is released and the translated Braille codes are loaded into the buffer, all 4 cells display “START” indicating it is time for reading. By pressing the “next” button (Fig.17a), the translated Braille codes are shown in Fig.19. The last 4 “CAPITAL” indicate the “END” of the text.

Compared to the simulation results shown in Fig.20, the measurement exactly


4 “SPACE”s (00/00/00/00h, when reset is asserted)
4 “START” signals (17/17/17/17h after loading)
“CAPITAL”, “T”, “e”, “x” (01/1E/24/33h)
“t”, “SPACE”, “t”, “o” (1E/00/1E/26h)
“SPACE”, “CAPITAL”, “B”, “r” (00/01/28/2Eh)
“a”, “i”, “l”, “l” (20/18/2A/2Ah)
“e”, “SPACE”, “SPACE”, “SPACE” (24/00/00/00h)
4 “END” Signals (01/01/01/01, “CAPITAL”s)
Figure 19. Quad-LED Braille cells display of “Text to Braille”

matches the simulation, from reset all the way to “END”. Please note that reader*_out has 8 bits while the Braille cell only requires 6 bits. The two most significant bits (MSBs) of read*_out are not used. They are designed for redundancy.

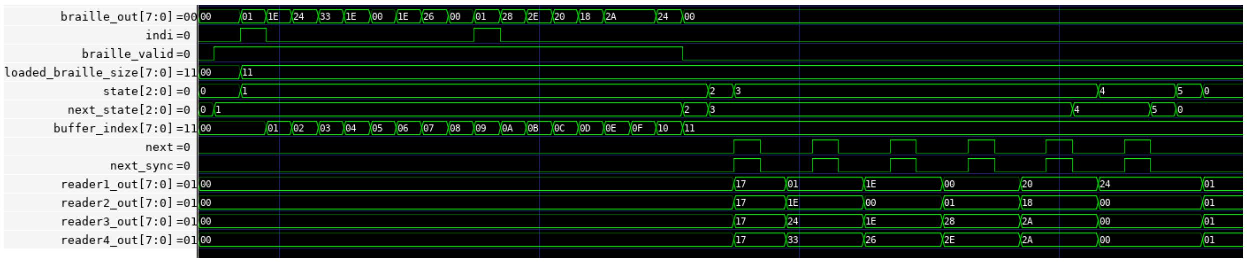


Figure 20. Simulation waveforms for the Quad-Braille-cell reader

5. Discussion: Comparisons and Future Improvements

A comparison of the proposed ASCII-to-Braille ASIC solution with previous hardware solutions is given in Table 1. The ASIC solution offers a more user-friendly, energy-efficient, cost-effective, mass-manufacturable, and space-conscious logic implementation in comparison to the solutions of FPGA, single-board computers, general-purpose microcontrollers, and discrete circuits.

Table 1. Comparison with previous hardware solutions

References	Zhang et al. (2006)	Kumari et al. (2020)	Saxena et al. (2022)	Proposed solution
Implementation method	FPGA	Single board computer	Discrete circuits + Single board computer	ASIC
Translation capability	Contracted Braille communication	Image-to-character	Upper-to-lowercase only	ASCII-to-character
Reading pace control	No	No	No	Yes
Translation speed	Fast	Slow	Medium	Fast
Area	Small	Large	Large	Small
Power Consumption	Medium	High	High	Low
Reliability	High	High	Low	High
Cost	High	High	High	Low

This ASIC's logic is easily scalable towards larger applications, as demonstrated in Section 4. While the Verilog hardware description language currently supports eight simultaneous readers, experiments #1 and #2 showcase the use of one and four readers, respectively. It is important to note that the latter experiments used fewer than eight readers due to pin limitations during the FPGA prototyping. By adjusting the number of readers, more efficient and practical Braille display applications can be developed. Considering that the average sentence length is 47.2 characters, a display with 64 Braille cells would provide more intuitive reading for visually impaired individuals.

However, there remain some challenges that need to be addressed before supporting 64 Braille cells. Implementing 64 readers in parallel would significantly increase the chip's width, resulting in a narrow rectangular shape instead of a square. This may introduce mechanical stress within the chip, potentially affecting the die saw process and mass production. Additionally, the varying routing distances connecting the 64 readers could result in timing skew, which may cause incorrect translations in a real-time operation. Both of these issues require careful investigation and solutions.

Moreover, due to limitations in the chip area and pin count as well as the absence of static random access memory (SRAM) in the Tiny Tape-Out program, the chip's memory is implemented using register-based read-only memory (ROM). The contents of this ROM are fixed and cannot be updated in real time. So, to create a practical product, the ROM would need to be replaced with SRAM to store an entire page of content. Additionally, an interface must be investigated and incorporated to bridge the connection between a computer and the converter, enabling real-time updates to the SRAM content.

6. Conclusion

This paper presents a novel ASCII-to-Braille conversion application specific integrated circuit designed for both electrical and mechanical output applications. The ASIC has demonstrated effective functionality and has successfully met the specific requirements of the intended application. With an inherently scalable design logic, expansion towards larger systems of multiple Braille cells could be easily implemented, allowing for more versatile and practical use cases. This chip provides a more user-friendly interface through the “next” pin that permits readers to proceed at their own pace as they interact with digital content. This work not only provides a practical solution for real-time text-to-Braille translation but also lays the groundwork for future developments in accessible technology.

Verilog source code for the ASIC can be viewed at *https://github.com/rileyguu/ASCII-to-Braille.git*

Acknowledgements

The author thanks MIT Beaver Works Summer Institute ASIC program for the technical and financial support to this work and chip fabrication through open-source Tiny Tape-out program 8 (TT-8).

References

Blenkhorn, P. (1997). A System for Converting Print into Braille. *IEEE Transactions on Rehabilitation Engineering, vol.5, no. 2, June 1997*

Digilent (2020). ARTY-Z7 schematic

Durre I., and Tuttle D. (1991). A Universal Computer Braille Code for Literary and Scientific Texts. *International Technology Conference, December 1991*. IEEE

Editors of Encyclopedia Britannica. (2019). Braille | writing system. In Encyclopedia Britannica. https://www.britannica.com/topic/Braille-writing-system

IIC-JKU. (2022). GitHub - iic-jku/IIC-OSIC-TOOLS: IIC-OSIC-TOOLS is an all-in-one Docker image for SKY130/GF180/IHP130-based analog and digital chip design. AMD64 and ARM64 are natively supported. GitHub. https://github.com/iic-jku/IIC-OSIC-TOOLS

Kumari, S., et al. (2020). Enhanced Braille Display: Use of OCR and Solenoid to Improve Text to Braille Conversion. *2020 International Conference for Emerging Technology (INCET)*. IEEE

Nelson, V. et al, (1995). Digital Logic Circuit Analysis and Design. Prentice Hall

Palnitkar, S. (1996). Verilog-HDL: A Guide to Digital Design and Synthesis. SunSoft Press, A Prentice Hall Title

Saxena, A., et al. (2022). A Device for Automatic Conversion of Speech to Text and Braille for Visually and Hearing Impaired Persons. *2022 8th International Conference on Signal Processing and Communication (ICSC*). IEEE.

Slaby W. (1990). Computerized Braille Translation. *Journal of Microcomputer Application, vol.13, issue n2, pp 107-113, 1990.* IEEE

Xilinx (July 2, 2018). Zynq-7000 SoC Data Sheet: Overview

Zhang, X., et al. (2006). Text-to-Braille Translator in a Chip. *2006 4th International Conference on Electrical and Computer Engineering (ICECE)*. IEEE.

Zhang, X., et al. (2007). A System for Fast Text-to-Braille Translation Based on FPGAs. *2007 3rd Southern Conference on Programmable Logic*. IEEE.