Morse Code to Text Translator

Nathaniel Van Devender, Michael Hardesty, and Dr. Rohit Dua
Undergraduate Student / Undergraduate Student / Associate Teaching Professor, Electrical and Computer Engineering, Missouri University of Science and Technology

Abstract
Previously, automated Morse Code translators were designed using a variety of programming languages, ranging from Machine Code on a Z80 Microprocessor to C++ on an Arduino. When looking at systems that are created on generalized processors, however, the question can be raised as to whether a non-generalized machine can be created to perform the task specifically, as this can lead to smaller form factors in design, faster operation speeds, and other such benefits. Using the fundamentals of Digital Logic Design, as well as some basic cryptography, several experimental circuits were constructed, and even the Morse Code was reconstructed to make this project work. In the end, most of the complications of the design stemmed from Morse Code itself – or, more specifically, how people in modern times use Morse Code. As a communication style, Morse Code has grown to become more of a shorthand than a full-scale code, and therefore could be far too complicated for any one device to decode accurately. As a training tool, however, the Morse Code to Text Translator (MCTT) will work well to help teach basic Morse Code structure, before setting its students free to learn more advanced Morse Code on their own.

Keywords
Morse code, Translator, FPGA, Interactive, Undergraduate Student Poster

Introduction

Morse Code has been an integral part of the communication industry for nearly two hundred years. Throughout this time, translation from Morse Code to English has been performed by human operators, many of whom would have required years of training to achieve fluency. Until recently, this would have been the only way to train Morse Code. Today, however, in the Silicon Age, we have access to components and knowledge that allow for computerization and automation of what once took man years to perfect – including, of course, the translation of Morse Code. While this would be a relatively simple matter to do in a high-level programming code, the question is raised as to whether the device can be constructed on the gate level directly, using concepts learned in Digital Logic classes.

Research and Results

The MCTT underwent three main stages – the Code Design stage, the Machine Design stage, and the LCD Interfacing stage. The Machine Design stage consists of three substages, during which three separate MCTT designs were built and tested. Each of these designs were drawn in the Intel Quartus II Web Edition v13.0sp1, and were hardware tested on an Altera DE2 FPGA running the Cyclone II EP2C35F672C6 chipset. The final display is programmed to an Altera DE0 FPGA running the Cyclone III EP3C16F484C6N chipset.

Stage 1 – Code Design

Morse Code Basics

Morse code is comprised of three characters: a "dit," a "dah," and a space. With these three characters, Morse code implements some relatively strict timing rules:

- A dit is the shortest measurement of time.
- A dah is the length of three dits.
- The space between dits and dahs within a letter is the length of one dit.
- The space between letters in a word is the length of three dits.
- The space between words is the length of seven dits.

Code Construction

The coding style created to convert Morse Code to binary underwent several major variations. The main issue with most of the variations was that the code was not easy to parse computationally. While watching lectures online for Digital Communications [1], the concept of prefix-free codes was raised, which inadvertently solved the problem with the Morse Code to binary conversion.

To construct the code, a binary "10" was determined as a "dit," a binary "1110" as a dah, and a binary "0" as a space. Each letter was padded to a length of 20 bits to help generalize the code. Not only is this code computer readable, but it also inherently follows the Morse timing rules.

A	00000000000010111000	**O**	00000011101110111000

Table 1 - Some Morse Code to 20-Bit Binary Examples

Stage 2 – Machine Design

Variable Clock Divider

The onboard clock on an Altera DE2 is 50MHz. While that is applicable for most computations, most humans are not able to input Morse code at a speed of nearly 2.5 million letters per second. To ensure human usability, the 50MHz clock was slowed internally using an array of divide-by-two circuits. This slow clock signal was then attached to a 16-bit counter. As the internal clock counts up on the counter, the user can select how many counter outputs are used to make up a single clock pulse, thus giving the user the ability to fine-tune the clock speed as desired.

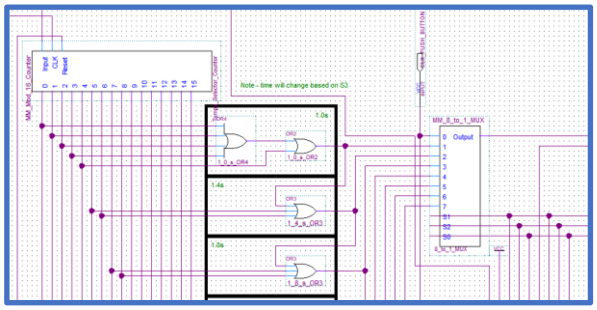


Figure 1 - Portion of Clock Speed Selection Circuit

Morse Code Bruteforce Decoder

This circuit is composed of three main sections. The first section is a 20-Bit SIPO shift register made of DFFs, as they were with the first circuit variation. This SIPO shift register accepts the incoming data from the user's button press, 20 bits at a time.

The second section of the bruteforce circuit is the "bruteforce" part – 26 separate sub-circuits that exist solely to look for specific codes in binary that translate to letters, as shown in Table 1.

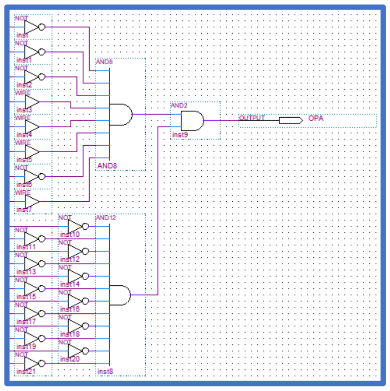


Figure 2 - Bruteforce Logic for "A"

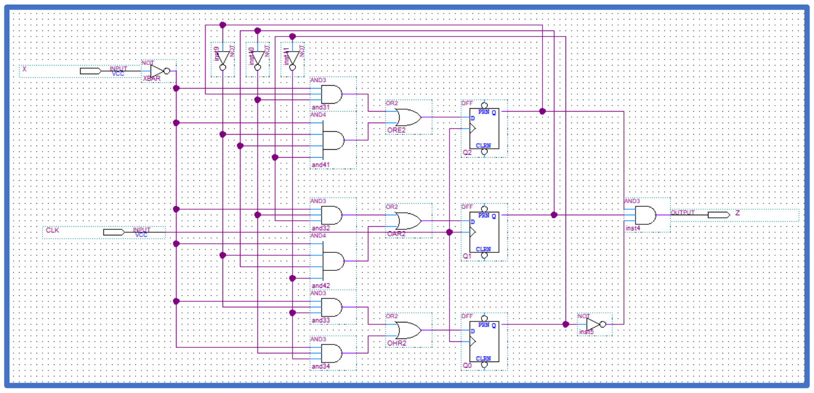


Figure 3 - 7-bit State Machine

The third section of the circuit handles the seven "dit" delay between words. This sub-circuit is a state machine that counts up from 0 to 6, and if no user input is detected, pulses a binary "1." All 27 outputs – A through Z and space – are connected to an LCD interface circuit.

Stage 3 – LCD Interface

LCD Module Control

The onboard driver for the LCD Module was found to include a CGROM, programmed with every character needed for the MCTT, and then some.

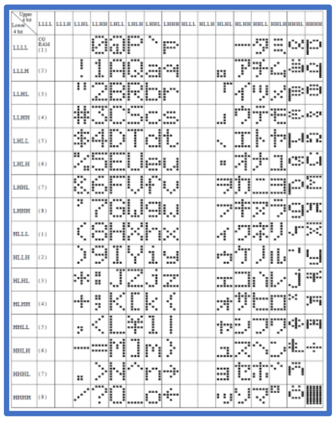
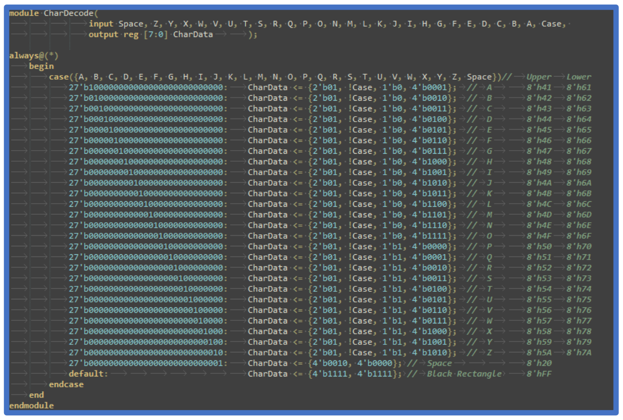


Figure 4 – CGROM Characters



```
module CharDecode(
            input Space, Z, Y, X, W, V, U, T, S, R, Q, P, O, N, M, L, K, J, I, H, G, F, E, D, C, B, A, Case,
            output reg [7:0] CharData       );

always@(*)
    begin
        case({A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, Space})//   Upper   Lower
            27'b100000000000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0001}; // A     8'h41   8'h61
            27'b010000000000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0010}; // B     8'h42   8'h62
            27'b001000000000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0011}; // C     8'h43   8'h63
            27'b000100000000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0100}; // D     8'h44   8'h64
            27'b000010000000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0101}; // E     8'h45   8'h65
            27'b000001000000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0110}; // F     8'h46   8'h66
            27'b000000100000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b0111}; // G     8'h47   8'h67
            27'b000000010000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1000}; // H     8'h48   8'h68
            27'b000000001000000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1001}; // I     8'h49   8'h69
            27'b000000000100000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1010}; // J     8'h4A   8'h6A
            27'b000000000010000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1011}; // K     8'h4B   8'h6B
            27'b000000000001000000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1100}; // L     8'h4C   8'h6C
            27'b000000000000100000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1101}; // M     8'h4D   8'h6D
            27'b000000000000010000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1110}; // N     8'h4E   8'h6E
            27'b000000000000001000000000000:   CharData <= {2'b01, !Case, 1'b0, 4'b1111}; // O     8'h4F   8'h6F
            27'b000000000000000100000000000:   CharData <= {2'b01, !Case, 1'b1, 4'b0000}; // P     8'h50   8'h70
            27'b000000000000000010000000000:   CharData <= {2'b01, !Case, 1'b1, 4'b0001}; // Q     8'h51   8'h71
            27'b000000000000000001000000000:   CharData <= {2'b01, !Case, 1'b1, 4'b0010}; // R     8'h52   8'h72
            27'b000000000000000000100000000:   CharData <= {2'b01, !Case, 1'b1, 4'b0011}; // S     8'h53   8'h73
            27'b000000000000000000010000000:   CharData <= {2'b01, !Case, 1'b1, 4'b0100}; // T     8'h54   8'h74
            27'b000000000000000000001000000:   CharData <= {2'b01, !Case, 1'b1, 4'b0101}; // U     8'h55   8'h75
            27'b000000000000000000000100000:   CharData <= {2'b01, !Case, 1'b1, 4'b0110}; // V     8'h56   8'h76
            27'b000000000000000000000010000:   CharData <= {2'b01, !Case, 1'b1, 4'b0111}; // W     8'h57   8'h77
            27'b000000000000000000000001000:   CharData <= {2'b01, !Case, 1'b1, 4'b1000}; // X     8'h58   8'h78
            27'b000000000000000000000000100:   CharData <= {2'b01, !Case, 1'b1, 4'b1001}; // Y     8'h59   8'h79
            27'b000000000000000000000000010:   CharData <= {2'b01, !Case, 1'b1, 4'b1010}; // Z     8'h5A   8'h7A
            27'b000000000000000000000000001:   CharData <= {4'b0010, 4'b0000}; // Space            8'h20
            default:                           CharData <= {4'b1111, 4'b1111}; // Black Rectangle  8'hFF
        endcase
    end
endmodule
```

Figure 5 - CharDecode.v

A demo project for the Altera DE2 was found to contain code for interfacing with the LCD module. This demo project, in conjunction with the data sheet, made reverse engineering the LCD CGROM possible. Using Verilog, several files were created that enabled interfacing the Morse Code Bruteforce Decoder circuit with the LCD Module. The CharDecode.v file, shown in Figure 5, identifies which letter is triggered and releases a corresponding 8-bit code to the LCD_Puppeteer.v file, which references the CGROM and sends the correct letter on the LCD. A third file, CacheRegister.v, was designed to continuously scan and update the LCD screen.

Conclusion

Morse Code, at first glance, seemed to be a simple code to deconstruct and translate via machinery. In the end, this proved untrue – the various rules and the subtlety of the timing structure of Morse Code adds just enough complexity to every step of the translation process to keep a project such as this interesting. There exist even more subtleties that have not been addressed. Some of these would be simple to incorporate into this machine, such as adding numbers or special characters. Others, such as personal callsigns, shorthand notation, or slight personal variations on typing speed, would be difficult to include on a hardware level. In the end, this machine acts as a useful personal trainer for an increasingly forgotten cryptographic language and will forever stand as a labor of love.

Acknowledgements
Michael would like to express his deepest gratitude to Dr. Rohit Dua, who has been a constant mentor and friend throughout the construction of this device.

Nathaniel is extremely grateful to his wife and daughter for putting up with “the insane ramblings of a madman,” as well as to Dr. Rohit Dua for his ever-present mentorship.

References

[1] MIT OpenCourseWare, "Lec 1 | MIT 6.450 Principles of Digital Communications I, Fall 2006," 28 April 2009. [Online]. Available: https://tinyurl.com/4wsauunp

Nathaniel Van Devender will graduate in 2024 with a BS in Electrical Engineering. He plans to work in the Electrical Power industry for his career but favors a myriad of other disciplines as hobbies. He has been overheard saying “the day I stop learning is the day that I die.”

Michael Hardesty is a first-generation college student and will be graduating in spring of 2024 with a BS in Electrical Engineering, a Computer Engineering minor, and an Automation minor. He is currently working at John Deere Reman - Electronics and plans to stay on the electronics side of the industry for his career. His hobbies are typically electronic in nature, but he can be found tinkering with pretty much anything. He has been overheard telling students how cool Nate is, but not to tell Nate because that would make things “weird.”

Rohit Dua, Ph.D., is an Associate Teaching Professor in the Department of Electrical and Computer Engineering at the Missouri University of Science and Technology and Missouri State University’s Cooperative Engineering Program. His research interests include engineering education. (http://web.mst.edu/~rdua/)