Hamming Codes

Computer Architecture Final Project, Fall 2022

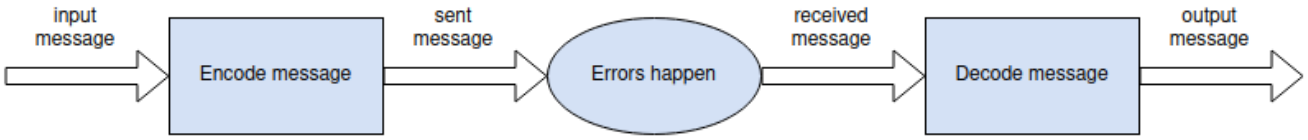
Lilo Heinrich and Emma Mack

Goal

Our goal was to breadboard a hamming encoder and decoder demo with LEDs to visualize the correct message, the corrupted message, and the corrected message. We created a working Hamming (7,4) encoder and decoder circuit where you can flip switches to create a 4-bit message, introduce a 1 bit error, and that bit error will be corrected in the final output message. An FPGA handles all of the logic.

How Hamming Codes Work

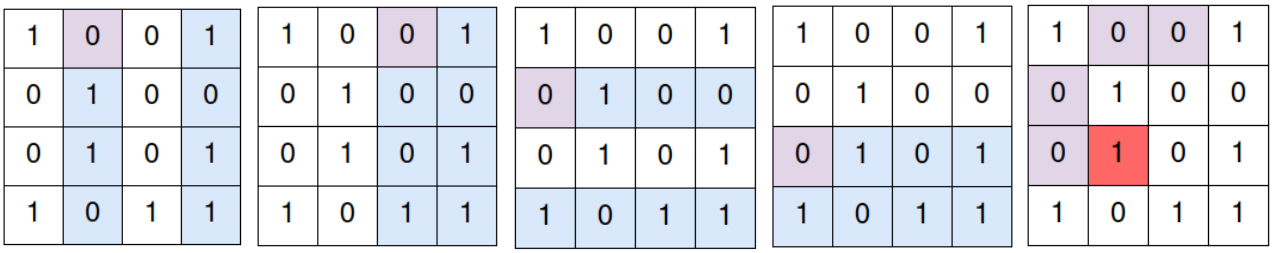
Data can be lost during transmissions over noisy communication channels or corrupted such as when a CD gets scratched. In order to still be able to obtain the correct original message, we can use error correction codes. The way an error correction code works is that there is some message you want to send, you encode it using an error correction code, and send it out. Once it gets received there may have been some noise which caused errors but after decoding you can recover the original message.



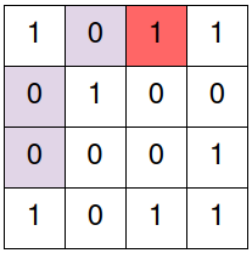
The principle of error correction is that only a subset of all possible messages are valid messages, so when you receive a message with an error you can identify that it is not valid and correct it to turn it back into a valid message. Error correction codes can never guarantee that your received message is the same as the original message. For example, random noise could turn a valid message into another valid message. Instead we can create error correction schemes that are robust up to a certain number of errors.

Hamming codes are a simple and elegant error correction code which can correct up to a 1 bit error anywhere in the message by using parity bits. Parity bits encode whether a group of bits added up to be an even number. If the bits add up to be even the parity bit is 0, if the bits add up to be odd the parity bit is 1. Having only one parity bit in a 16-bit block of data for example can tell us if an error occurred but cannot tell us where. However if there were two errors (or any even number of errors) inside the message the parity bit would not indicate any error. One parity check on its own is weak, so what we can do is apply parity checks to certain subsets of a message. If we choose these subsets carefully we can use them to figure out the location of any single-bit error, even an error in the parity bits.

Take the example of a 16-bit block and suppose that the bit in purple is a parity bit on the group of bits in blue for each of the following:



To figure out which bit is wrong let's first look at the parity bits to check the columns. Looking at parity bit (1), there is an odd number of bits equaling one, yet an even parity bit, so something within those two columns is wrong. Looking at row parity bit (2), there is an even number of bits equaling one and a parity value of 0, meaning everything in those two columns is correct, narrowing our error to the second column from the left. Moving on to rows, parity bit (3) is correct, eliminating rows two and four from errors. Lastly, parity bit (4) is incorrect, narrowing our error finally to the third row and second column. Even if a parity bit is wrong, that can also be detected and corrected by this algorithm; you can work through this example to confirm if you wish:

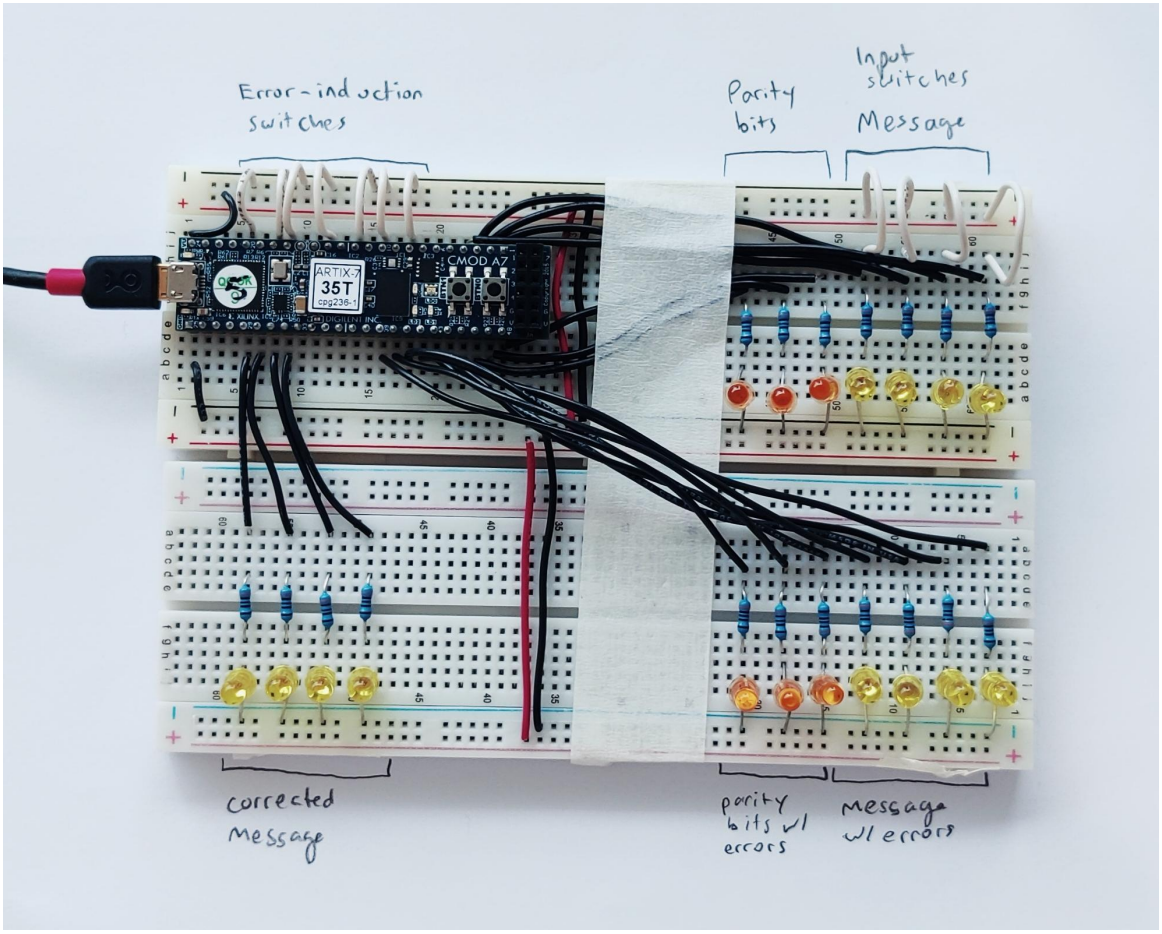


This is a Hamming (15,11) code meaning that there are 15 total bits and 11 message bits. The reason it is not (16,12) is that the bit at the top left corner cannot be used for a message bit. If all of the parity bits are zero, we would not know if this meant the data was error-free or if that bit at the top left corner was wrong. However, it can be used for an overall parity bit which can tell us whether a second error occurred but not where it occurred, which is called an extended Hamming code.

Essentially a Hamming code is using these parity bits to perform a binary search for the error in a message, and it can be expanded to any power of two. The example above used $2^4 = 16$ bits, with 4 parity bits and 11 message bits. The table below shows Hamming code sizes, illustrating how as the message size increases the number of parity bits is the logarithm of total bits so it is an efficient encoding scheme. However, it is still only single fault tolerant and typically the longer a message is the more errors are likely to occur.

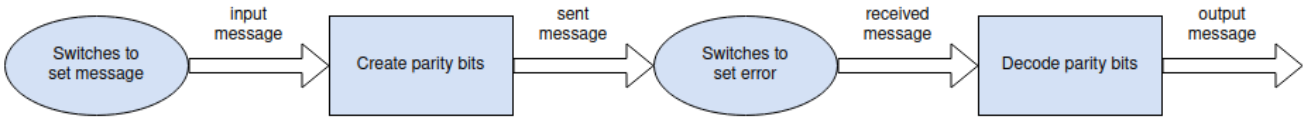
Power	Total bits	Parity bits	Message bits	Hamming (n,m)
3	$2^3 = 8$	3	8-3-1 = 4	(7,4)
4	$2^4 = 16$	4	16-4-1 = 11	(15,11)
5	$2^5 = 32$	5	32-5-1 = 26	(31,26)
6	$2^6 = 64$	6	64-6-1= 57	(63,57)
7	$2^7 = 128$	7	128-7-1=120	(127,120)
8	$2^8 = 256$	8	256-8-1 = 247	(255,247)

What We Implemented



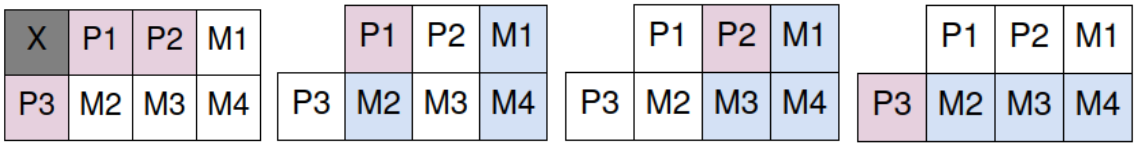
Hamming code breadboard encoder and decoder

We programmed the hardware logic for a Hamming (7,4) encoder and decoder. To show that it works we have switches to set the message and LEDs to visualize the message. Here is the process of steps:



1. Set a message (upper right) by setting the input switches to PWR or GND. The yellow LEDs will light up with your message, and the orange LEDs will light up with the calculated parity bits.
2. Induce one error (upper left) by setting one of the error-induction switches to PWR.
3. See the message with the induced error on the lower right: one of the bits will be flipped as compared to the original set message + parity bits.
4. See the corrected message on the lower left. If the error corrector is working, it should be the same as the original message, despite induced errors.

We chose (7,4) because it is a manageable size to build the circuit. Here are the parity groups:



Layout Parity bit 1 Parity bit 2 Parity bit 3

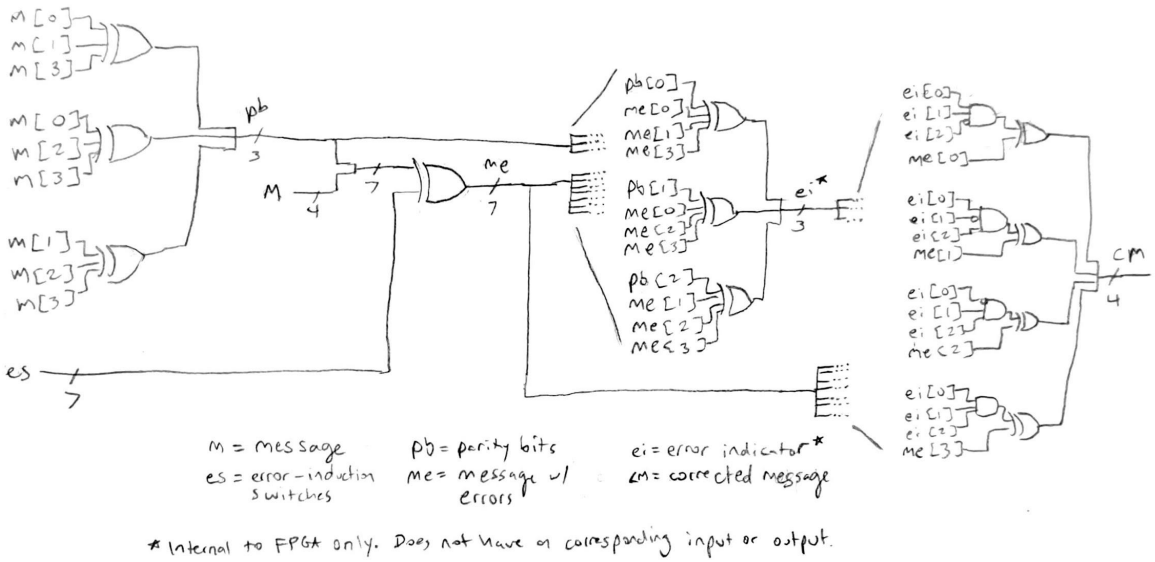
The parity groups can also be shown as a table. To calculate the parity bits, we simply XOR together all three of the input bits which gives a 1 when the inputs add up to be odd and a 0 otherwise.

To decode a message, we first determine whether the parity bit's value matches the parity of the group by again XORing together the input bits along with the parity bit. This gives a logical output of 0 if the parity bit matched the input bits, and a 1 if there was an error somewhere within that group of all four bits.

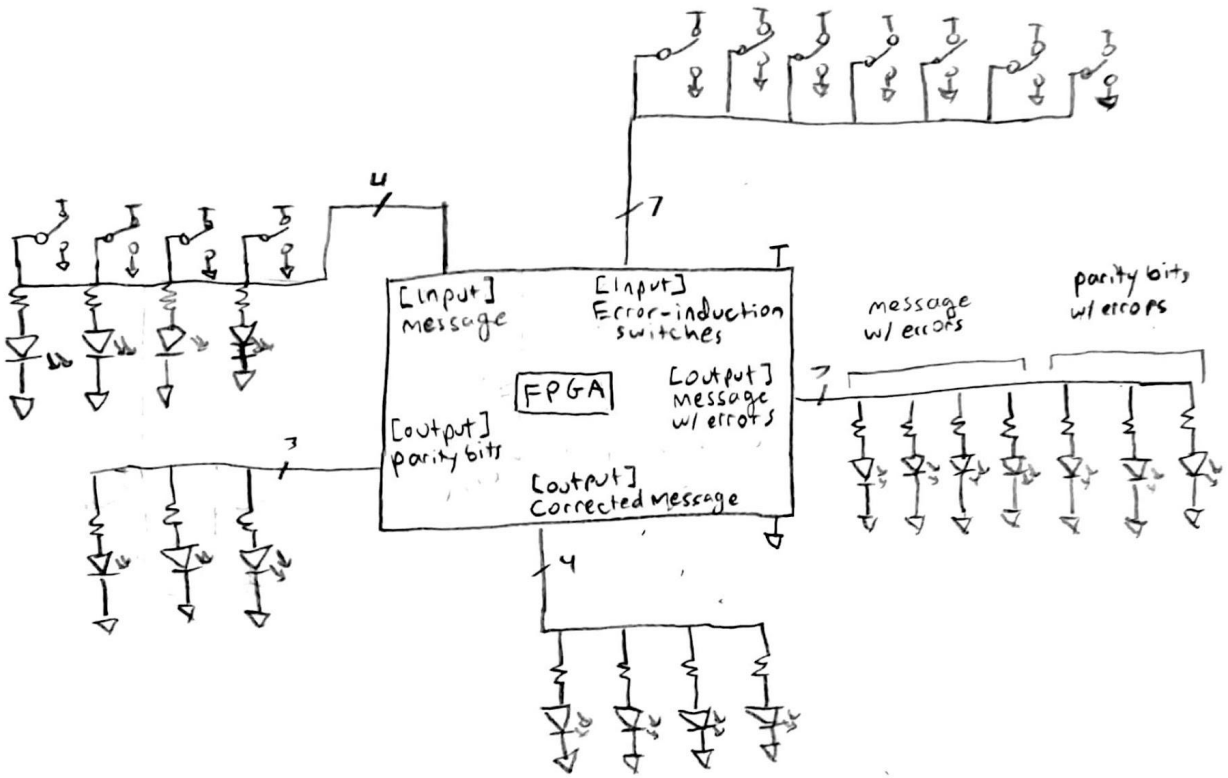
To calculate the correct message, we need to know for each bit if it had the error, and if so, to flip it. For example, imagine the two parity bits P1 and P2 both indicated an error while the parity bit P3 did not indicate an error. The only common bit between parity groups P1 and P2 that is not in group P3 is M1, so message bit M1 has the error. The way to implement this with logic gates is simply to AND together the error indication for P1, P2, and NOT P3. We can generalize this for the other bits in the message. To calculate whether M2 has an error, AND together P1, NOT P2, and P3 to target just M2. Finally, for each message bit, we XOR together the value of this error check with the value of that message bit, effectively flipping back the errored bit (if there is one).

M4	M3	M2	M1	P3	P2	P1
x		x	x			x
x	x		x		x	
x	x	x		x		

Schematics



Schematic of logic internal to the FPGA



Schematic of all elements external to the FPGA

Appendix

Code

All code used for this project can be found here. The main file with Hamming Code logic is main.sv.

Resources

Project idea and walkthrough: Ben Eater, "What is error correction? Hamming codes in hardware"
Hamming codes theory: 3Blue1Brown - "How to send a self-correcting message (Hamming codes)"

Demo Video

View our demo video.