**Error Correction Encoding / Decoding (1 page)** For each error correction code you consider include the its characteristics and justify your choice in the context of your project. For example: code rate, code length, minimum Hamming distance, memory, etc

**What is Hamming code correction ½?**

The error correction code that we decided to implement in our design is the ALTECC IP core included within the Quartus library. This is a type of forward error correction which means that the receiver can detect and correct errors without having to send communication back to the sender. The coding scheme used to encode the data is the Hamming Code scheme. Hamming encoding is a method of attaching redundant bits to the data in order to tell when an error has occurred. It is more sophisticated than a simple parity check but it applies the same idea. In a simple parity check, a parity bit is appended to the end of a string of bits and is either 0 or 1 depending on the even or oddness of the bits. If the previous bits have an odd number of ones, this bit will be set to 1. If the previous bits have an even number of ones, then it will be set to 0. In this sense, if the receiver receives an odd number of ones it immediately knows that an error has occurred. This breaks down when there is more than a single bit error in the encoded message. However, the receiver does not know where the error has occurred and has no way to correct the message. Hamming codes are a way of placing parity bits strategically throughout the message so that the receiver can identify that and error has occurred and also where it has occurred. This is done by placing the parity bits at the powers of 2 throughout the message. So if the message is treated as a square block of bits, parity bits would be placed at 1, 2, 4, 8, 16, and so forth along the columns and the rows. Then, by a simple check of the parity corresponding to the sectioning off of the columns and rows based on these locations, a single bit error can be located anywhere in the message. This scales so that the number of redundant bits attached to large messages is very low. This is referred to as the code rate.

**Why did we choose it, what are the advantages compared to other techniques ½?**

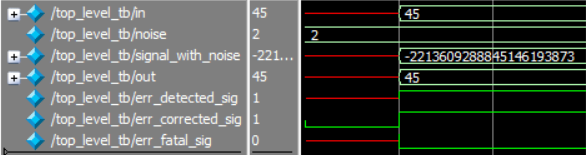
We chose this method of error correction for a number of reasons. First, it does not require communication back to the transmitter and can correct errors automatically. Next, it is suitable for the kinds of errors we expect to see in the channel as a consequence of noise. We don’t expect large burst errors to occur where many bits are altered. We expect small distributed errors where noise spikes can alter certain parts of our data. Throughput and an efficient code rate were critical to meeting the various design goals. We could not afford to send a large number of redundant bits through the channel and sticking to a simple encoding technique kept things manageable. We are using a Hamming (72, 64) code. This means that a 64-bit message is sent to the encoder. The encoder takes this message, attaches 8 parity bits, and sends the 72-bit encoded message across the channel. 7 of these bits allow for correction of single-bit errors and the 8th bit placed at location zero allows us to detect 2-bit errors although we cannot correct them. This gives us a code rate of 0.11 or 11% redundant bits transmitted over the channel. The ALTECC core is also entirely combinational which yields minimal latency and only requires some buffering before and after the blocks to have appropriate bus widths as inputs

**Error Correction Encoding / Decoding**

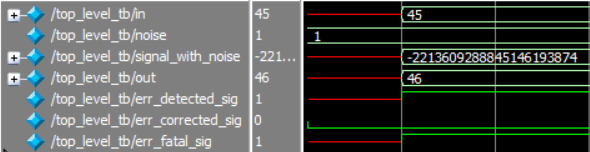
To verify the Error Correction Encoding/Decoding process, we first started with a very simple testbench to ensure that data could be reliably encoded, sent across the channel, and then be decoded at the other end. After this was confirmed, the next step was to introduce noise into the channel and see how much corrupted data could be restored. For the Hamming (72, 64) encoding we are using, the decoder can correct 1-bit errors and detect 2-bit errors although it cannot correct them. The relevant flag signals are

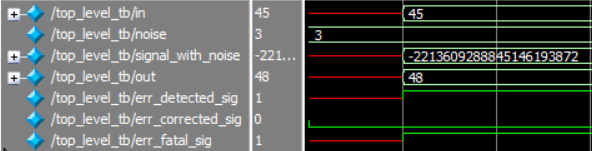
* *err\_detected\_sig* indicates that an error has been detected.
* *err\_corrected\_sig* says a single-bit error has been found and corrected.
* *err\_fatal\_sig* denotes that a double-bit error has been found, but not corrected. You must not use the data if this signal is asserted.

To do this, we will look at sending the value 45 across the channel which has a binary representation of 101101. Noise will have a value of 2 which will be added to the signal.



47 in binary is 101111 which is only a 1-bit difference from our original signal





Individual Contributions:

* Error Correction Encoding / Decoding Simulink:

For the first part of the project design, I worked on the modelling of the Error Correction Modules in Simulink. As this was the start of the project, we had not yet decided upon what method would best suit the design goals and requirements that we had to meet. First, I tried to implement the algorithm manually by using a simple parity bit check per byte. This was one of the first methods suggested to us in the course and seemed like a good and simple method. I tried building up the algorithm using logic blocks and math functions in Simulink and made a fair amount of progress. However, I then discovered pre-existing blocks for error encoding existing within Simulink and this immediately became clear as the better alternative. The methods I looked at specifically were Hamming Codes, BCH Codes, and Reed Solomon Codes. After some experimenting and research into the methods I concluded that Reed Solomon would be a good choice for our design. I built a simple test environment with a Random Integer Generator, Reed Solomon encoder/decoder, a Gain block to introduced noise into the channel, and an Error Rate Calculation Block. Next, I experimented with inputs and noise levels to see what kind of error correction that we could accomplish with the parameters that we had chosen. I chose N = 256, K = 223 as our parameters. This meant that we would have 32 redundant bits of data or a Code Rate of 12.5%. With these parameters we would be able to correct up to 16 symbol errors anywhere in our message. This seemed like an acceptable code rate and the error correction appeared very good although I would later discover that we could meet our design goals with a method that can correct far fewer errors.

* Error Correction Encoding / Decoding FPGA:

After Demo 1, we shifted our focus onto FPGA implementation. I began work on the error correcting mechanism and started thinking about what additional challenges would be present on the physical board. I started with the intention of using Reed Solomon encoding again as our mechanism of error correction. The Quartus library includes two IP Cores relating to Reed Solomon encoding. The one I began work on was Reed Solomon II. I did a fair amount of testing and ensured that the timing of the input signals was correct in order to encode the data. I generated both modules and began simple testing in ModelSim. Eventually I found out that I would need a license in order to synthesize these modules in Quartus. There seemed to be a path to getting a license for free but I then discovered another option which was the ALTECC IP Core. This was another Core found within the Quartus library but this did not require a license. It used a Hamming scheme to encode the data and was not a black box like Reed Solomon II. I could actually peer in and see the algorithm which helped me a lot in understanding how the module was functioning. I decided to use a Hamming (72, 64) scheme. This gave us a Code rate of 11% which was suitable for our transmission of data. The block was also combinational which yielded minimal latency. I constructed a simple test system and simulated in ModelSim while introducing different amounts of noise into the channel. After I confirmed that single-bit errors could be effectively corrected, I moved into integrating these blocks into the overall system. This required buffering at the input of the encoder and the output of the decoder in order to meet data rates of the system. I did this in conjunction with Ilia who added them to the overall system and adjusted them such that they met the required rates of data transmission.

Team Effectiveness:

* What we did well:

I think our team performed effectively because we all shared the common goal of striving to do well in this course and create a design that we were proud of. Our group involved people with various situations. Some on co-op, some taking multiple courses, and some in different time-zones. However, we were able to perform well as a cohesive unit because everyone was on the same page with regards to what we sought to achieve.

* Suggestions for Improvement:

There is always room for improvement. If I was to start the course again from Day 1, I think our team would have benefited from each member being more actively involved in each other’s activities. We always collaborated and brought everything together as a whole, however there were certain specific aspects of each person’s tasks that truly only that person understood on a fundamental level. Division of labour is important but I believe we would have benefited from more active communication with the specific details of each person’s tasks.

* Changes since the beginning:

I think one thing all of us learned since Module 1 is the importance and usefulness of Scrum meetings. At the beginning of the course, we conducted these more as a formality and to submit the required weekly scrum info. As the course progressed, I think all of began to see the value in scheduled weekly meetings. We maintained active communication over discord and met over voice chat on many occasions. However, having the additional weekly meetings at the same time where we would exchange information proved highly useful. It is a practice I intend to continue on projects I undertake in the future.

Other Comments:

* Breaking up test cases into small chunks rather than trying to simulate the top-level module from the start is a good way to ensure the modules are working properly. If they aren’t, it’s easier to pinpoint the problem.
* Maintaining a weekly schedule and setting goals and milestones. The break between Demo 1 and Demo 2 was large. It was important to maintain a steady workflow such that we didn’t have an insurmountable amount of work at the end.
* Active communication with team-members. Communication on a daily basis was critical to our success.
* Come to TUT’s and Office Hours with questions. The feedback in the Office hours in the second half of the course proved very useful.
* I learned a lot about Error Encoding. I learned a lot about communication systems as a whole. However, error correction was the task I was dedicated to and so most of my time was devoted there. I found the concept that you could allocate a small number of redundant bits and correct errors over a large array of data bits really cool. The important differences between the methods was also interesting and this is an area of digital design I would like to do more of in the future.