Optimization of Multi-Channel BCH Error Decoding for Common Cases

Russell Dill
Master's Thesis Defense – April 20, 2015
Bose-Chaudhuri-Hocquenghem (BCH)

- BCH is an Error Correcting Code (ECC) and is used to correct errors in noisy communications channels or storage mediums.
- Allowing for noise (errors) enables the use of much faster communications channels and much denser storage mediums.
- Used in:
  - Wireless communication links
  - NAND flash storage
  - Magnetic storage
  - On-chip cache memories
  - DRAM memory arrays
  - Data buses
BCH Code

- BCH is a configurable block based error correcting code (ECC)
  - Message is broken into fixed sized blocks and then each block is formed into a codeword
  - Redundant bits (ECC) are added to message to generate codeword
  - Size of codeword is configurable
  - Error correction capability is configurable
  - Redundant bits are used by receiver to detect and correct errors
Decoding is broken into 3 independent stages:

1. **Syndrome Vectors**
   - $S = (S_4, S_3, S_2, S_1)$

2. **Error Locator Equation**
   - $\lambda_2 x^2 + \lambda_1 x^1 + 1$

3. **Chien Search**
   - Find error locations

Diagram shows the flow from Codeword to Error locator polynomial coefficients to Serial data.
Syndrome Calculation

- Breaks down received codeword into a set of vectors that depend only on the error locations, easiest stage of decoding
- Syndromes evaluate to zero if no errors are present
- Accepts data serially, outputs syndrome vectors once complete

\[ \lambda_2 x^2 + \lambda_1 x^1 + 1 \]
Generate Error Locator Polynomial

- Uses the syndrome vectors to calculate a polynomial whose roots give the error locations
- Uses iterative algorithm known as Berlekamp-Massey
- Outputs coefficients when complete

\[ \lambda_2 x^2 + \lambda_1 x^1 + 1 \]
Factoring Error Locator Polynomial

- Uses brute force algorithm known as Chien search
- Roots give the locations of the errors
- Errors in message can now be corrected
- Outputs stream indicating error locations, 0 for no error, 1 for error
Multi-channel BCH Decoding

- Multi-channel decoders combine multiple decoders in parallel
  - 8 syndrome units, 8 error locator units, 8 root solving units
- Increases throughput
- Can be fed by multiple parallel communications or storage channels
- Can be fed by an interleaved code
  - Typically used in communications to spread error bursts across multiple blocks
Related Work

- BCH decoding is a heavily researched area.
- Invented in 1959, widely used today.
- Almost all of that research has focused on improving standalone encoders and decoders.
- Research has concentrated on improvements to both efficiency and throughput.
Existing Work to Improve Throughput

- Bit-parallel operation
  - Previous BCH decoders operated 1 bit at a time
  - Best demonstrated by Hwang (1991)
  - Syndrome unit calculates multiple input bits in parallel
  - Chien root finding unit calculates multiple output bits in parallel
  - Intermediate stages require no modification as they pass data as a unit in a single clock cycle

- Advantage – Flexible, easy to apply to existing designs
- Disadvantage – Lowers clock rates, increases logic complexity
Existing Work to Improve Throughput

- Multi-channel operation
  - Abraham et al. (2010) shows an example in flash memory storage system
  - Shi et al. (2004) shows an example of interleaving in communications links
- Advantage – Scales all properties linearly
- Disadvantage – Requires modification of design, requiring either multiple input channels or an interleaved code
Syndrome Efficiency Improvements

- Lin and Costello (1983) have demonstrated a mathematical relation between syndrome vectors.
  - Relation can be used to calculate only a subset of the syndromes, and then perform a simple expansion step to recover the rest.
  - Some syndromes can be calculated by a more efficient method (Lin and Costello, 1983).
- Bit-parallel optimizations for LFSR can be applied to syndrome computation (Pahri, 2004).
  - Most common syndrome computation method is LFSR.
  - Bit-parallel operation is common to increase throughput.
Error Locator Improvements

Jamro (1997) shows how the number of Berlekamp-Massey iterations can be reduced by intelligently loading the initial state. Jamro also observes the necessity to multiply 3 factors and shows a more efficient solution by pairing 2 serial multipliers of different basis.

- One multiplier accepts parallel input, gives serial output.
- The other accepts serial input, gives parallel output.
- Can operate simultaneously if basis rearrangement is performed serially.
- Requires novel serial basis conversion circuit.
Chien Efficiency Improvements

- Bit-parallel Chien search improvements
  - Large bit-parallel Chien search circuits consume a large amount of decoder area and power
  - Chen and Parhi (2004) demonstrate a group matching scheme that can apply to Chien circuits and reduce logic complexity by 22%
  - Yang, Chen, and Chang (2013) demonstrate the relative cost of implementing multiplication serially or in parallel
Our Observation

- Most BCH decoder capability goes unused
  - Chance of entire decoder being used is 1/30 billion
- In multi-channel configuration, on average only 1/3 of decoders are required
  - Remaining blocks contain no errors
- Eventually, full decoder will still be required
- Presents great opportunity for improvement
Our Observation

Majority of decoder capacity goes unused (on average)

Questions?
Our Approach

- Apply ideas to multi-channel decoder
  - Allow possibility that not enough hardware is available **right now** (leads to performance penalty)
  - We still include at least 1 full decoder
  - Resize remainder of decoders
- Syndromes (S) indicate error vs no error
  - All channels must undergo syndrome, but we can eliminate later stages
- Include arbitrator to select first available unit
Our Approach

- Can we do better?
- After error locator (Σ), we know the error count
- Single error blocks can be solved directly since they are of the form: \( \lambda_1 x^1 + \lambda_0 \)
- Create reduced root locator (r) to replace expensive Chien units (C)
- Add second arbitrator to select correct type of root solver based on error count
Implementing the Reduction in Hardware Blocks

- Large percentage of blocks contain zero errors
- Calculate the probability that a block contains zero errors based on BER, $p$, and block size, $n$:

$$P_0(n) = (1 - p)^n$$

- Choose number of units such that there is only a small chance (miss rate) that insufficient hardware is available
- In example on right, 3 error locator units are chosen as there is a less than 2% chance of 4 or more blocks containing errors
Choosing the Acceptable Performance Penalty

- Miss rate is probability that at any given time, insufficient hardware is available.
- Miss rate is chosen based on trade-off between area savings and performance penalty.
- Same equations that determine probability of a certain number of errors within a block can be used to calculate the probability of a certain number of blocks containing errors.
- For our experiments, 2% was chosen as a good balance.
  - Most gains are seen before the 2% mark.
Choosing the Number of Error Locator Units

- For each number of possible units, 1…channel count, plot probability that more than that count will be required.
- Find smallest count $m$ below the 2% threshold.
- Implement that many error locator units ($\Sigma$).
- Number of units required increases with targeted Bit Error Rate (BER). 
  - For BER of $5 \cdot 10^{-6}$, only 1 unit is required.
  - For BER of $1 \cdot 10^{-4}$, only 5 units are required.
Architecture of Pooled Decoder

- Pooling is used to connect to a full set of inputs to a reduced number of units
- Pooled decoder inserts arbitrators between stages
- Allows data to flow to first available decoding unit
- In case of root solver (C/r), allows arbitrator to choose unit type based on error count
- Still requires full set of syndrome units (small overhead)
Beyond Removing Units, Optimizing Units

- Handling blocks with no errors allowed us to remove entire units
- Blocks containing 1 error are still a common case
- Error count is only known after error locator polynomial step (Σ)
- To take advantage of this observation, we need to create special reduced root solvers (r)
- Error polynomial will be of the form $\lambda_1 x^1 + \lambda_0$
  - Full Chien requires: $\lambda x^n \ldots \lambda_3 x^3 + \lambda_2 x^2 + \lambda_1 x^1 + \lambda_0$
- Direct solution with simple algebra
Reduced Root Solver

- Solve and Simplify
  - Negation is a null operation
  - $\lambda_0$ is always 1
- Provides direct, one step method to find root of error locator polynomial in the case of 1 error
- However, the solution cannot be used to directly give an integer index to the error location because BCH codes are computed using an algebra known as finite fields
- We need to learn a little about finite fields
Finite Field Arithmetic

- A finite field contains a finite set of elements, operations on any two elements produces another element in the field
- Elements can be represented in different forms
  - Computation typically uses the binary representation of the polynomial form
- All operations are performed modulo the generator polynomial, which defines the field
- Moving from the polynomial form to the power form (index) is $O(n)$ where $n$ is the number of elements in the field (Typically in the thousands).

$$x^3 + x + 1 \text{ over } GF(2^3)$$

<table>
<thead>
<tr>
<th>Power form</th>
<th>Polynomial form</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>$x^1$</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>$x^2$</td>
<td>$x^2$</td>
<td>0010</td>
</tr>
<tr>
<td>$x^3$</td>
<td>$x + 1$</td>
<td>0100</td>
</tr>
<tr>
<td>$x^4$</td>
<td>$x^2 + x$</td>
<td>0110</td>
</tr>
<tr>
<td>$x^5$</td>
<td>$x^2 + x + 1$</td>
<td>0111</td>
</tr>
<tr>
<td>$x^6$</td>
<td>$x^2 + 1$</td>
<td>1011</td>
</tr>
</tbody>
</table>

Very costly
Finite Field Addition

- Polynomials are added similarly to normal algebra:
  \[(x^2 + x) + (x + 1) = x^2 + 2x + 1\]
- But the coefficient of each term is taken modulo the characteristic of the field, which for binary fields is 2, \(GF(2^n)\):

\[x^2 + 2x + 1 = x^2 + 0x + 1 = x^2 + 1\]
- This is the same as taking the XOR of the binary representation of the polynomial form.
- Addition is easy in finite fields!
- Subtraction is defined to be the same as addition in finite fields (Negation is a null operation).
Finite Field Multiplication

- To multiply two elements in power form, just add the exponents modulo the size of the field:
  \[ x^3 + x^4 = x^{(3+4)\mod 7} = x^0 \]

- Multiplication in polynomial form is performed similarly to normal algebra, but taken modulo the generator polynomial:
  \[ (x + 1)(x^2 + x) = x^3 + 2x^2 + x = x^3 + x \]

- And then to take it modulo the generator polynomial, we subtract it out:
  \[ (x^3 + x) - (x^3 + x + 1) = 1 \]
Reduced Root Solver

- Observation – we still need to cycle through each bit in the error output (Decoder streams error locations serially)
- Rework equation again

- Load register with $\lambda_1$
- Multiply by $x^1$ each cycle
- When register contains 1, we have multiplied by the correct power of $x$
- We have counted the correct number of cycles and have reached the root/error location
- Implement with LFSR, extends cheaply to multi-bit support

<table>
<thead>
<tr>
<th>Power form</th>
<th>Polynomial form</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^0$</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>$x^1$</td>
<td>$x$</td>
<td>0010</td>
</tr>
<tr>
<td>$x^2$</td>
<td>$x^2$</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td>$x + 1$</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td>$x^2 + x$</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>$x^2 + x + 1$</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td>$x^3 + 1$</td>
<td>1010</td>
</tr>
</tbody>
</table>
Usage of Linear Feedback Shift Register (LFSR)

- Multiplies element by $x^1$, looping through the table
- Operates on binary representation
- Step 1: shift elements one to the left
- Step 2: subtract generator polynomial if needed

LFSR can be modified with input/output
- Multiply two values serially
- Divide two values serially, producing a quotient and remainder
- Used for many BCH operations

Generator polynomial

$\begin{array}{c|c|c}
\text{Power form} & \text{Polynomial form} & \text{Binary representation} \\
\hline
0 & 0 & 0000 \\
1 & 1 & 0001 \\
x & x & 0100 \\
x^2 & x^2 & 1001 \\
x + 1 & x + 1 & 0011 \\
x^2 + x & x^2 + x & 1010 \\
x^2 + x + 1 & x^2 + x + 1 & 1011 \\
\end{array}$
Comparison with Chien Search

- Brute force method of finding roots
- Loads registers with coefficients ($\lambda_n$) at first cycle (mux used to select)
-Multiplies by $\alpha$ (constant) each subsequent cycle
  - $n$ indicates coefficient index
- $t$ (number of errors that can be corrected) blocks are required per channel, which is also the number of coefficients
- Requires $t$ registers, muxes, and parallel multipliers. Expensive
- Sums output of all blocks and compares result with 0 to detect roots

Portion of Chien root finder, duplicated $t$ times
Comparison with Chien Search

- Bit-parallel operation is scales at a rate beyond linear because of long delays and fanout requiring register duplication.
- Diagram demonstrates hardware necessary to operate on 8 bits in parallel.
- Parallel implementation of reduced root solver only requires 1 LFSR.
  - Output of LFSR is compared against 8 constants per cycle.
  - LFSR is advanced 8 steps per cycle.

9 multipliers
2 multiplexers
2 registers
Choosing the Correct Number of Root Solvers

- Similar to calculation of number of error locator units required
- Instead we look at blocks with more than 1 error (instead of 1 or more)
  - Blocks with 1 error can be served by reduced root solver
  - Blocks with more errors need Chien search
- Calculate probability that more than \( m \) Chien units will be required
- Plot and choose minimum number of units below 2% threshold
Experimental Setup

- Implemented pipelined BCH decoding architecture in Verilog (baseline)
- Target Virtex-6 FPGA, 200MHz timing for comparisons
- Create pooled architecture with arbitrators
- Make parameters compile time configurable for testing many configurations
- Run Place & Route of design for the target, examine results to determine area for the chosen parameters
Area Optimized Decoder Results

- Results of applying our ideas to a set of multi-channel BCH decoder configurations
- We allow a 2% performance impact
- We gain
  - 47%-71% area savings
  - 44%-59% dynamic power savings
Increasing Throughput

- Fit more powerful optimized decoder in area of baseline decoder
- Find highest bit-parallel level of optimized decoder that fits within area of baseline decoder
  - Each channel within baseline decoder operates on 4 bits
  - Increasing bit-parallel capability of each channel increases throughput, but grows area
  - Optimized version of decoder gives us plenty of headroom to grow!
- 300%-500% increase in throughput!
How Multi-Bit Increases Throughput

- Computing error locator polynomial does not need to be modified for multi-bit support
- Syndrome computation streams input data, limits throughput
  - Syndrome computation is typically performed via LFSR, which scales well to bit-parallel operation
- Root solver streams error locations as output, limits throughput
  - Increasing Chien search to bit-parallel operation involves duplicating multipliers, muxes and registers
  - Including many multipliers lead to timing problems requiring register duplication, resulting in even more area
  - Our reduced root solver is simply an LFSR and extends well to bit-parallel operation
Extending Flash Lifetime

- Optimization savings allow us to implement a more powerful decoder in the same area.
- Find optimized decoder with largest targeted Bit Error Rate (BER) that fits into the area of the baseline decoder.
- We can now support an increased lifetime for flash memory, whose error rate increases as it ages.
- We are able to achieve a 1.4x-4.5x increase in flash lifetime.

<table>
<thead>
<tr>
<th>Original BER</th>
<th>Original t</th>
<th>Optimized BER</th>
<th>Optimized t</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 × 10^{-6}</td>
<td>5</td>
<td>1.0 × 10^{-4}</td>
<td>10</td>
</tr>
<tr>
<td>2 × 10^{-5}</td>
<td>7</td>
<td>1.2 × 10^{-4}</td>
<td>11</td>
</tr>
<tr>
<td>5 × 10^{-5}</td>
<td>8</td>
<td>1.5 × 10^{-4}</td>
<td>12</td>
</tr>
<tr>
<td>1 × 10^{-4}</td>
<td>10</td>
<td>2.0 × 10^{-4}</td>
<td>13</td>
</tr>
</tbody>
</table>
Flash Lifetime Background

- P/E cycles cause flash memory to wear
- Decreasing process sizes cause wear to happen sooner
- Flash manufacturers recommend an ECC strength in bits to reach a specified flash lifetime
- ECC strength chosen is a trade-off based on decoder requirements
- By providing more ECC strength in a same sized decoder, we can change that trade-off
- Cai et al. (2012) shows age/BER generally follows the relation:

\[
\frac{BER_2}{BER_1} = \left[ \frac{age_2}{age_1} \right]^2
\]
Conclusions

- We examined possibilities for improving multi-channel BCH decoders.
- By allowing a 2% performance degradation, we experienced massive gains, 47%-71% area savings, 44%-59% dynamic power savings.
- We can fit faster and more powerful optimized decoders in the same area of the baseline decoder.
  - $3x-5x$ increase in throughput
  - $1.4x-4.5x$ increase in flash lifetime
Questions?