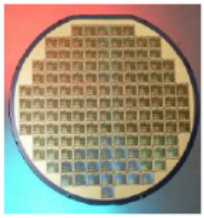
TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY



VLSI SYSTEMS RESEARCH CENTER

VLSI LABORATORY

DEPARTMENT OF ELECTRICAL ENGINEERING | DEPARTMENT OF COMPUTER SCIENCE

Controller for DNA-Based Data Storage

Final Report

Submitted by:

Jad Taya

Aslan Showgan

Supervised by:

Dr. Amit Berman

Table of Contents

Abstract

As digital information continues to accumulate, so does the demand for higher density and longer-term storage solutions. DNA offers an abundant, sustainable, and stable data storage solution, with storage density many orders of magnitude better than today's best methods. However, neither the synthesis, nor the amplification or the sequencing of DNA strands can be performed error-free today and for the foreseeable future. In order to make synthetic DNA available as digital data storage media, specifically tailored forward error correction schemes need to be applied. In this work we have implemented efficient and robust modulation and forward error-correcting schemes, both adapted to the DNA channel. The implementation is based on a model developed by a team from Technicolor Hannover & Harvard Medical School[1].

1. Introduction

By 2025, humans are expected to produce 175 zettabytes of data each year[2], and with the demand for digital data storage outpacing our storage capabilities[3], new methods for data storage are required. DNA data storage is a possible solution, it has density of petabytes of information per gram (see figure 1) and its durability is unmatched by conventional data storage methods[4] (see figure 2) while only needing to keep it in a cool and dry environment, meaning it is energy efficient[5]. DNA is also attractive because of how easy it is to copy using PCR (Polymerase Chain Reaction) and the fact that it will stay relevant forever (fields such as medicine will always be interested in sequencing DNA)[6].

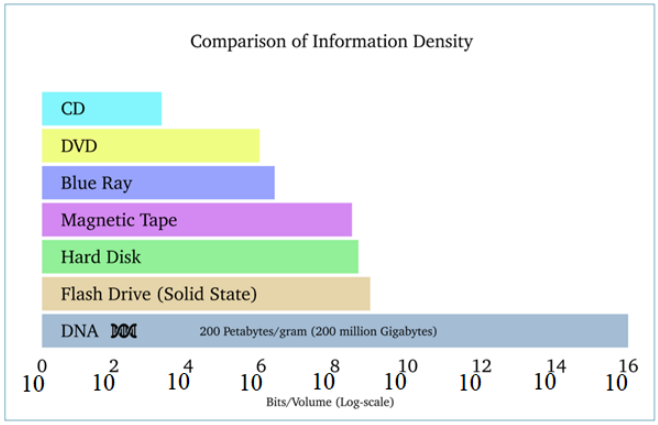


Figure 1: Comparison of information density between DNA based data storage and conventional data storage methods in LOG-scale. [7] [8] [9]

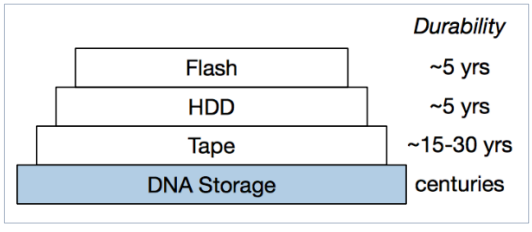


Figure 2: Comparison of durability between DNA based data storage and conventional data storage methods. [10]

Biological, bio-chemical and bio-physical processes are prone to errors, physical and chemical effects also cause errors over the course of time. Modern synthesizers can string together the four base nucleotides of DNA: Adenine (A), Cytosine (C), Guanine (G), Thymine (T) to create arbitrary strings of synthetic DNA. When synthesizers create DNA errors may occur, these errors can be divided into two categories, errors that are present in conventional means of digital data storage (such as swap errors), and errors that only materialize because of the characteristics of the DNA channel (such as run-length errors)[1]. We will deal with the first type using forward error correcting code and a unique modulation and with the second type using the modulation only, in the next section run-length errors will be presented, and the way to deal with them will be explained.

2. Modulation

In 2012 George M. Church and his team proved the concept of storing digital data in synthetic DNA[11], by representing 0 as either A or C and 1 as either G or T. This simple modulation meant that 1 bit was stored per nucleotide, a more efficient encoding scheme would allow us to store more than 1 bit per nucleotide, such as storing 2 bits per nucleotide[12] (00 for A, 01 for C, etc…), but this encoding technique does not meet the run-length constraint. Run-length limitations exist because current day synthesizers produce deletion errors at a high rate when the strand of DNA has a sequence of more than 3 identical nucleotides[13], in this work we will implement an encoding method that guarantees a maximum of 3 identical nucleotides in a sequence for any binary input[1].

The modulation maps 8 binary bits to 5 nucleotides with the following rules: using **Table 1**, bits 0 and 1 are mapped to the first nucleotide, bit 2 and 3 are mapped to the second nucleotide and bit 4 and 5 are mapped to the fourth nucleotide. Using **Table 2**, bits 6 and 7 are mapped to the third and fifth nucleotides. This produces 4 possible encodings, and to guarantee a maximum run-length of 3, they are subjected to the following constraints: the first three nucleotides shall not be the same and the two last nucleotides shall not be the same.

Value	Nucleotide
00	A
01	C
10	G
11	T

Table 1: Mapping of first 3 bit-tupel to 3 nucleotides

Value	Option 1	Option 2	Option 3	Option 4
00	AA	CC	GG	TT
01	AC	CG	GT	TA
10	AG	CT	GA	TC
11	AT	CA	GC	TG

Table 2: Mapping of last 2 bits to a pair of nucleotides

Figure 3: Tables 1 and 2 used in the modulation.[1]

This modulation produces a storage capacity of 1.6 bits per nucleotide and ensures that a single DNA swap error affects at most 2 bits during demodulation[1]. In conclusion, to store any array of bits in DNA, we first split the array into bytes and then apply the aforementioned modulation. For example, the byte 00011011 is mapped as follows: using **Table 1**, bits 0 and 1 are mapped to the first nucleotide A, bit 2 and 3 are mapped to the second nucleotide C and bit 4 and 5 are mapped to the fourth nucleotide G, so far we have AC_G_. Using **Table 2**, bits 6 and 7 are mapped to the third and fifth nucleotides, either AT, CA, GC or TG. This produces 4 possible encodings: ACAGT, ACCGA, ACGGC, or ACTGG. ACTGG violates the second constraint (the two last nucleotides shall not be the same) so we are left with 3 possible encodings: ACAGT, ACCGA or ACGGC.

3. Forward Error Correction Scheme

The team in [1] based their Forward Error Correction (FEC) scheme on a solely phenomenological error analysis of the experimental data Church and his team gathered [11]. In the experimental data they found that the swap error rate lies between $6.0 \cdot 10^{-4}$ and $1.4 \cdot 10^{-3}$, while insertion and deletion error rates lie between $1.0 \cdot 10^{-3}$ and $5.0 \cdot 10^{-3}$.

According to our modulation design, a single nucleotide error only lead to a maximum of two consecutive bit errors. Therefore, the data is effectively protected with a strong BCH code. In this work, a BCH (63, 39) code with a minimum Hamming distance of 9 was chosen to protect the data, where 39 are data bits [14]. The BCH code can detect at least 8 bit errors and correct up to 4 bit errors.

BCH (63, 39) has a field with the following parameters: $q = 2$ and $m = 6$ (therefore $n = 2^6 - 1 = 63$). The minimum Hamming distance is 9, therefore $d = 9$. For $GF\left(2^6\right) = GF(64)$, and based on the polynomial $x^6 + x + 1$ with primitive root $\alpha = x + 0$.

BCH code is constructed using polynomials over a Galois field, its set of valid code words consists of polynomials that are divisible by a given fixed polynomial called the generator polynomial. Let $m_i(x)$ satisfy $m_i\left(\alpha^i\right) \bmod \left(x^6 + x + 1\right) = 0$ then it's the i^{th} minimal polynomial, the generator polynomial of the BCH code is defined as the least common multiple $g(x) = lcm\left(m_1(x), \ldots, m_{d-1}(x)\right)$.

The first 8 minimal polynomials are:

$$m_1(x) = m_2(x) = m_4(x) = m_8(x) = x^6 + x + 1$$

$$m_3(x) = m_6(x) = x^6 + x^4 + x^2 + x + 1$$

$$m_5(x) = x^6 + x^5 + x^2 + x + 1$$

$$m_7(x) = x^6 + x^3 + 1$$

$$\begin{aligned} g(x) = lcm\left(m_1(x), \ldots, m_8(x)\right) &= m_1(x) \cdot m_3(x) \cdot m_5(x) \cdot m_7(x) \\ &= x^{24} + x^{23} + x^{22} + x^{20} + x^{19} + x^{17} + x^{16} + x^{13} + x^{10} \\ &\quad + x^9 + x^8 + x^6 + x^5 + x^4 + x^2 + x + 1 \end{aligned}$$

3.1 Encoding

Because any polynomial that is divisible by the generator polynomial is a valid BCH codeword, BCH encoding is merely the process of finding some polynomial that has the generator polynomial as a factor. So, our goal is to transmit a codeword $s(x)$ such that:

$$s(x) = q(x)g(x)$$

This encoding method leverages the fact that subtracting the remainder from a dividend results in a multiple of the divisor. Hence, if we take our message $p(x)$ as before and multiply it by $x^{n-k} = x^{63-39} = x^{24}$ (to "shift" the message out of the way of the remainder), we can then use Euclidean division of polynomials to yield:

$$p(x)x^{24} = q(x)g(x) + r(x)$$
$$\Rightarrow s(x) = q(x)g(x) = p(x)x^{24} - r(x)$$

3.2 Decoding

To decode the received message, implement the following algorithm:

1. Calculate the syndromes for the received code and determine the number of errors t and the error locator polynomial from the syndromes, if it is zero then no corrections are required.
2. Calculate the roots of the error location polynomial
3. Find the error locations and correct the errors by swapping the bits where errors are located.

3.2.1 Syndrome Calculation

The received code is the sum of the correct codeword and an unknown error code. The syndrome values are formed by considering the received code as a polynomial and evaluating it at $\alpha, \alpha^2, \ldots, \alpha^8$, since they are the zeros of the generator polynomial, of which the codeword is a multiple, they are also zeros for the codeword. Examining the syndrome values thus isolates the error vector so one can begin to solve for it.

Thus, the syndromes are:

$$s_i = \mathrm{Re}ceived\_code(\alpha^i) \text{ for } i = 1, 2, \ldots, 8$$

If the syndromes are all zero, then the decoding is done. Otherwise save the non-zero syndromes in an array.

3.2.2 Key Equation Solver

Initialize $r_0(x) = x^9$, $r_1(x) = Syndromes(x)$, $g_0(x) = 0$ and $g_1(x) = 1$,

Apply the Euclidean algorithm on the polynomial $r_0(x)$ and $r_1(x)$ in order to find the error locating polynomial:

1. Do a long division, divide $r_0(x)$ by $r_1(x)$ to get the quotient $q(x)$ and the remainder $r(x)$.
2. Update $g(x) = g_0(x) - q(x) \cdot g_1(x)$.
3. Check if the degree of $r(x)$ is less than max_errors (4) if true finish.
4. Update $r_0(x) = r_1(x)$, $r_1(x) = r(x)$, $g_0(x) = g_1(x)$ and $g_1(x) = g(x)$. Repeat step 1.

3.2.3 Find and Fix Errors

1. Calculate value $e_i = g(\alpha^i)$ for $1 \le i \le 63$.
2. Let $h(x) = \begin{cases} 0 & x = 0 \\ 1 & x \neq 0 \end{cases}$. Calculate $H(x) = \sum_{i=1}^{63} h(e_i) \cdot x^i$
3. Return the result of $s(x) + H(x)$ in $GF(2)$.

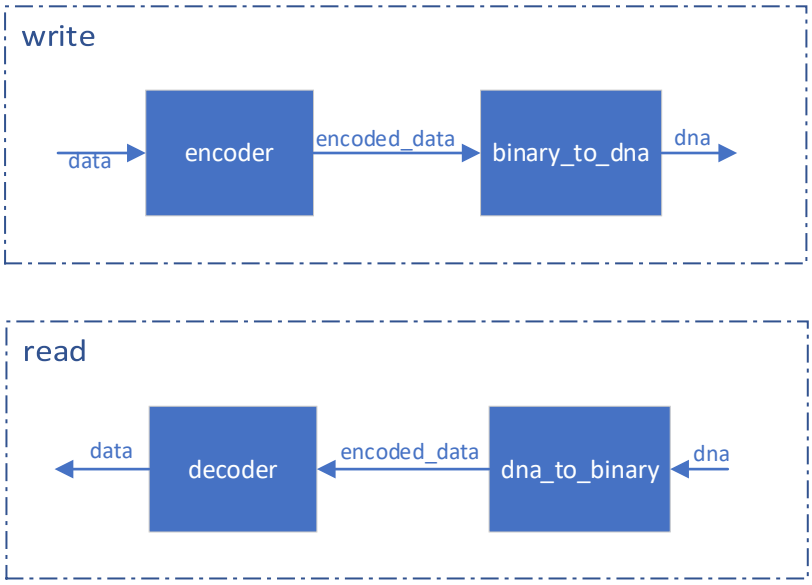
4. Software Simulation

Using Matlab, we simulated writing and reading ~10 Kbyte of data to and from DNA. The process is as follows, first we split the data into parts, each part containing 39 consecutive bits. For every such part:

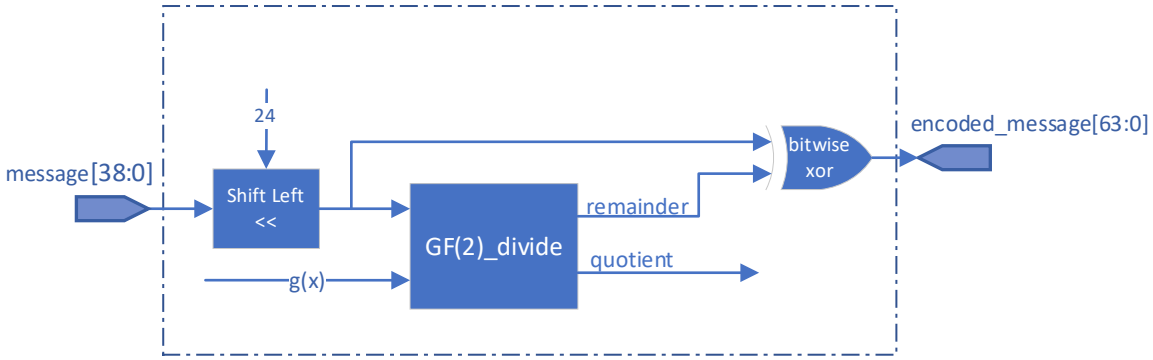
1. Encode every 39 consecutive bits of data by adding 24 EDC parity bits (BCH (63, 39)) to them, now there are a total of 63 bits.
2. Convert the 63 binary bits into 40 nucleotides, using the modulation scheme from section 2.
3. Simulate synthesizer errors by randomly swapping up to 2 nucleotides.
4. Convert the 40 nucleotides into 63 binary bits, using the modulation scheme from section 2 to demodulate the DNA string.
5. Decode the 63 bits - detect the errors and correct them and return the original 39 bits.

We also created a test vector for the hardware simulation, the test vector contains the data and the nucleotide errors, both of which are randomly generated. It also contains the corresponding nucleotide strings, to make sure that the hardware simulation returns the same results as the software simulation.

5. Architecture



5.1 Encoder

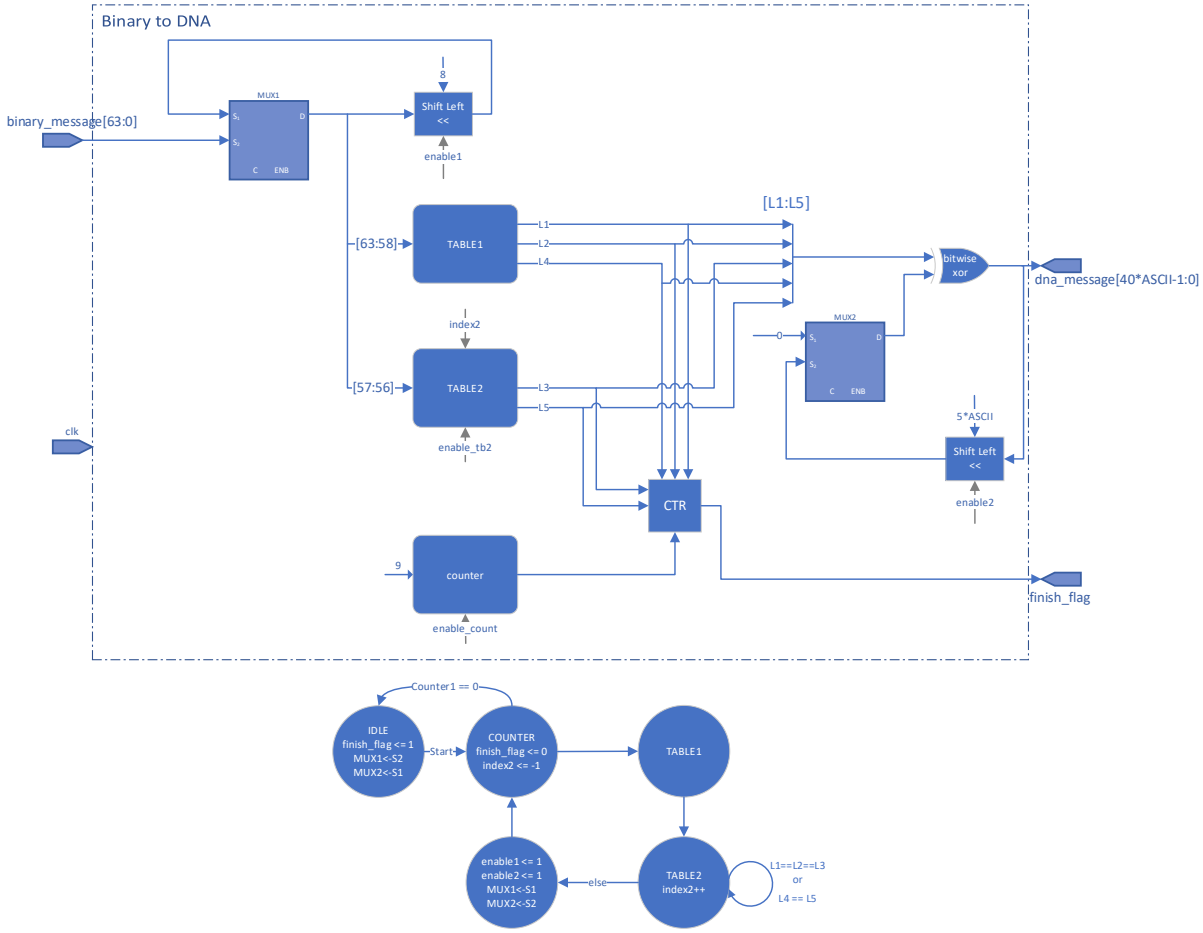


It receives a binary message of 39 bits and outputs a BCH encoding of the message (39 data bits + 24 ECC bits).

G(x) = 25'b1110110110010011101110111 is the generator polynomial of the BCH code.

We multiply the input by x^{24} and divide the result by the generator polynomial to receive the ECC bits as the remainder of the operation, we then add the ECC bits to the data bits.

5.2 Binary to DNA



Units TABLE1 and TABLE2 found in the next page are implemented according to the tables from section 2.

1. Initialize dna_message to be an empty string. Divide the 64 bit binary message into 8 sections of 8 bits (this is done by shifting the 64 bit message 8 bits to the left and taking the 8 MSB every iteration).
2. Map bits 8-2 of the current 8 bit section using table 1 to the appropriate nucleotides (8-7 -> L1) (6-5 -> L2) (4-3 -> L4).
3. option_index = 0.
4. Map the last 2 bits using table 2. Fetch the nucleotides using the option_index to receive one of the four options (2-1 -> L3 L5).
5. If the resulting 5 nucleotides do not satisfy the constraints from section 2, set option_index = option_index + 1 and from repeat step 4.
6. Shift the dna_message to the left by 5*ascii_size and append the 5 letters (L1…L5) to the end of the dna_message.
7. Repeat from step 2 until all 8 sections of the binary message have been mapped.

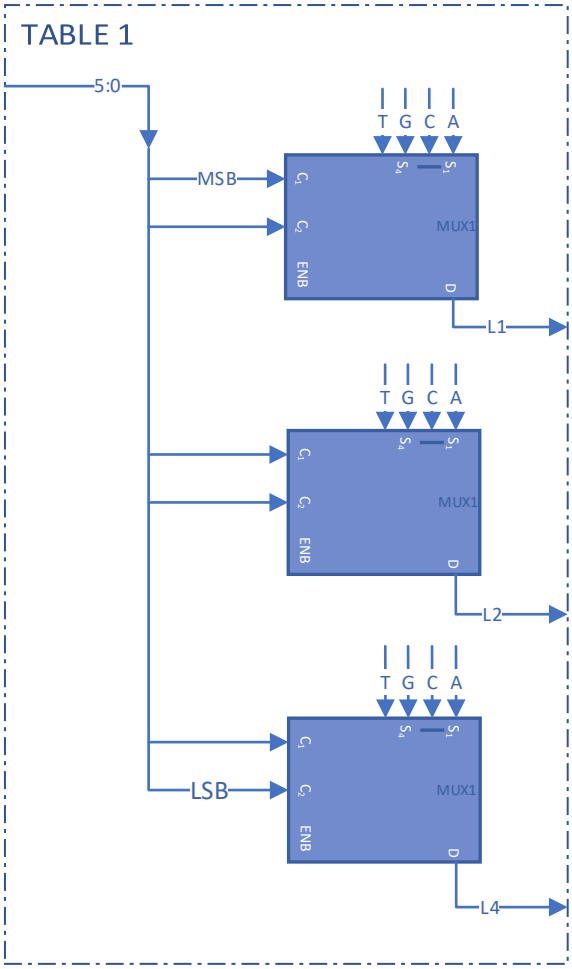


TABLE 1
5:0
T G C A
MSB
MUX1
L1
T G C A
MUX1
L2
T G C A
LSB
MUX1
L4

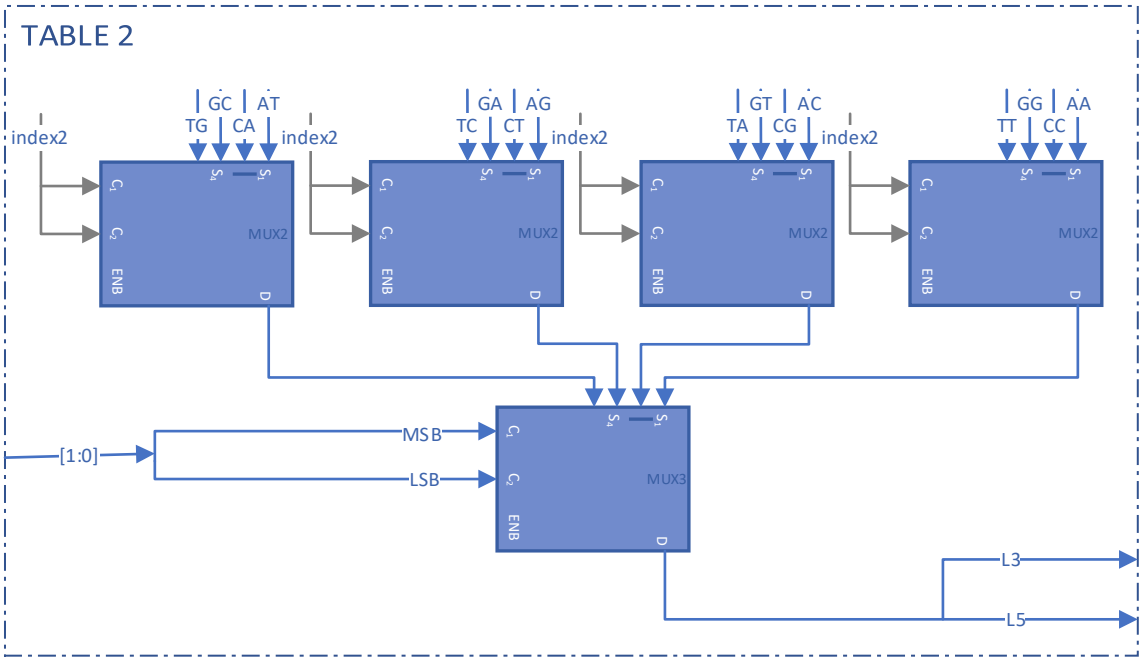
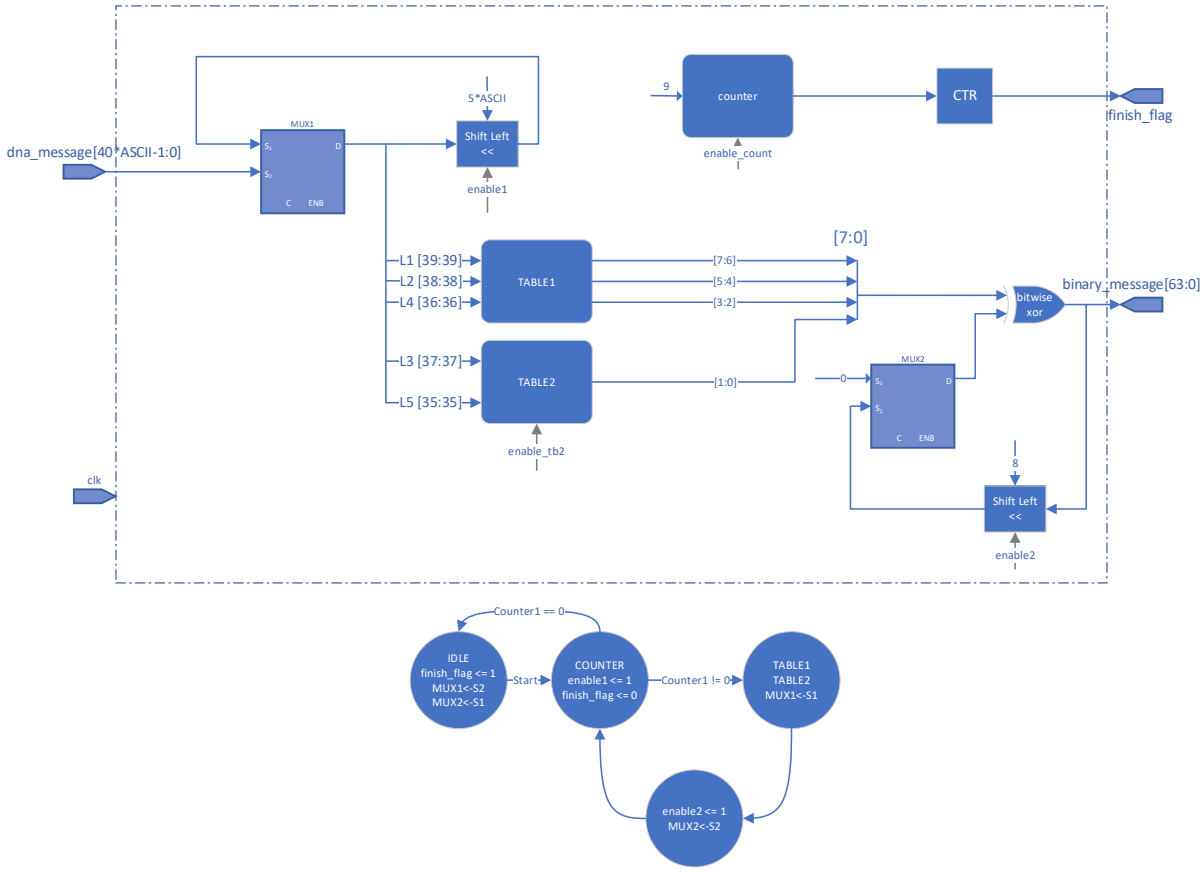


TABLE 2
index2
GC AT
TG CA
index2
GA AG
TC CT
index2
GT AC
TA CG
index2
GG AA
TT CC
MUX2
[1:0]
MSB
LSB
MUX3
L3
L5

5.3 DNA to Binary



Units TABLE1 and TABLE2 found in the next page are implemented according to the tables from section 2.

1. Initialize binary_message = 0. Divide the 40 nucleotides message into 8 sections of 5 nucleotides (this is done by shifting the 40 ASCII characters message 5 characters to the left and taking the 5 MSB characters every iteration).
2. Map nucleotides L1, L2 and L4 of the current 5 nucleotides section using table 1 to the appropriate bit pairs
   (L1 -> 8-7) (L2 -> 6-5) (L4 -> 4-3).
   Map the nucleotides L3 and L5 using table 2 to the appropriate bit pair (L3 L5 ->2-1).
3. Shift the binary_message to the left by 8 and append the 8 bits to the end of the binary_message.
4. Repeat from step 2 until all 8 sections of the DNA message have been mapped.

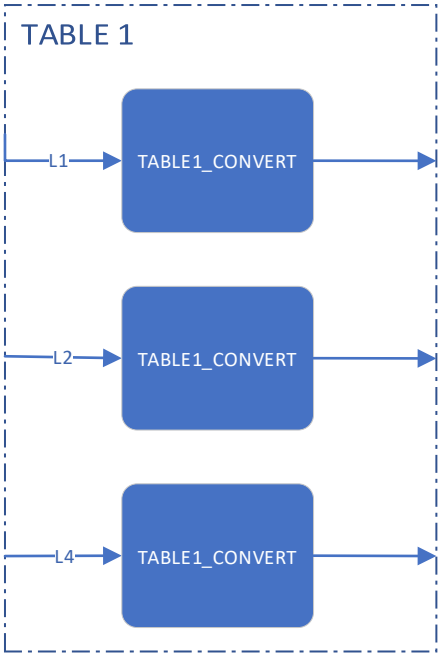


TABLE 1
L1
TABLE1_CONVERT
L2
TABLE1_CONVERT
L4
TABLE1_CONVERT

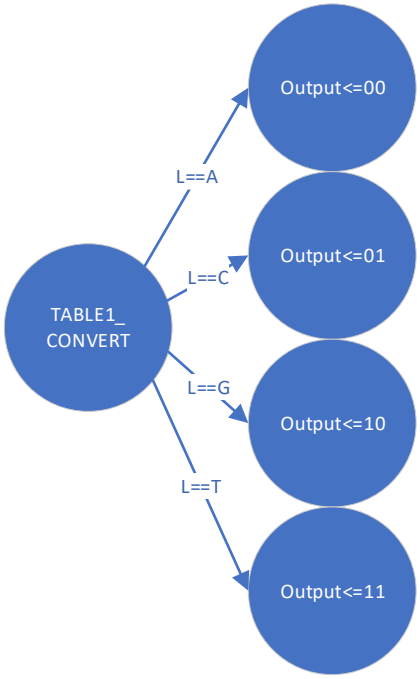


TABLE1_
CONVERT
L==A
L==C
L==G
L==T
Output<=00
Output<=01
Output<=10
Output<=11

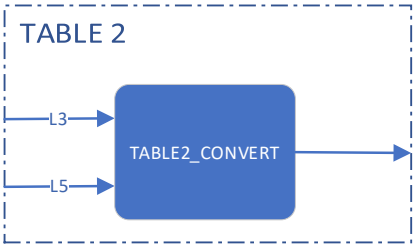


TABLE 2
L3
L5
TABLE2_CONVERT

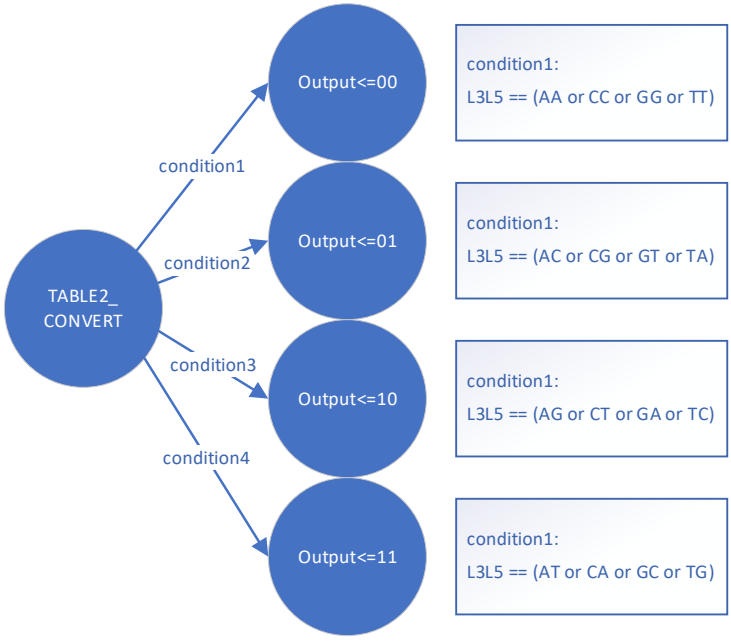
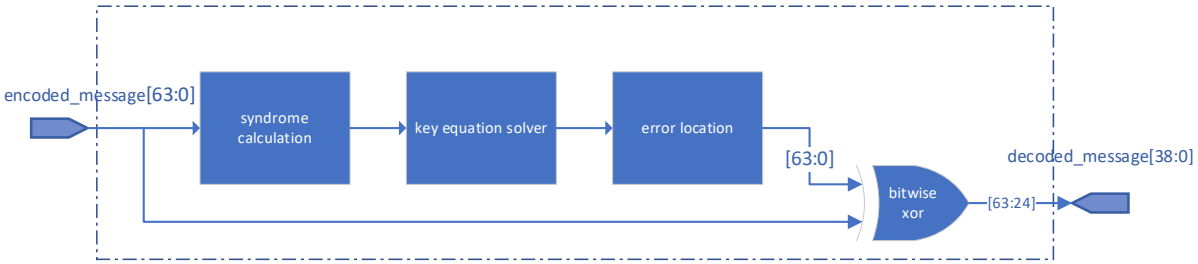
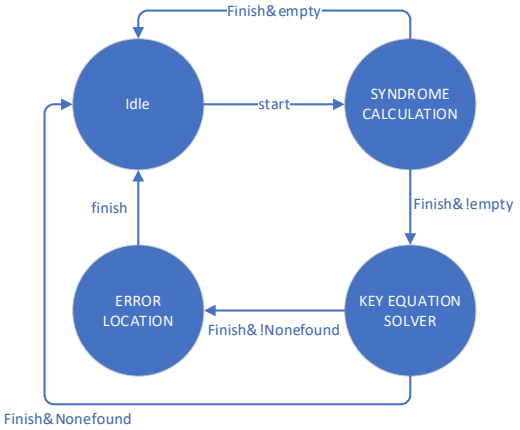


TABLE2_
CONVERT
condition1
condition2
condition3
condition4
Output<=00
Output<=01
Output<=10
Output<=11
condition1:
L3L5 == (AA or CC or GG or TT)
condition1:
L3L5 == (AC or CG or GT or TA)
condition1:
L3L5 == (AG or CT or GA or TC)
condition1:
L3L5 == (AT or CA or GC or TG)

5.4 Decoder

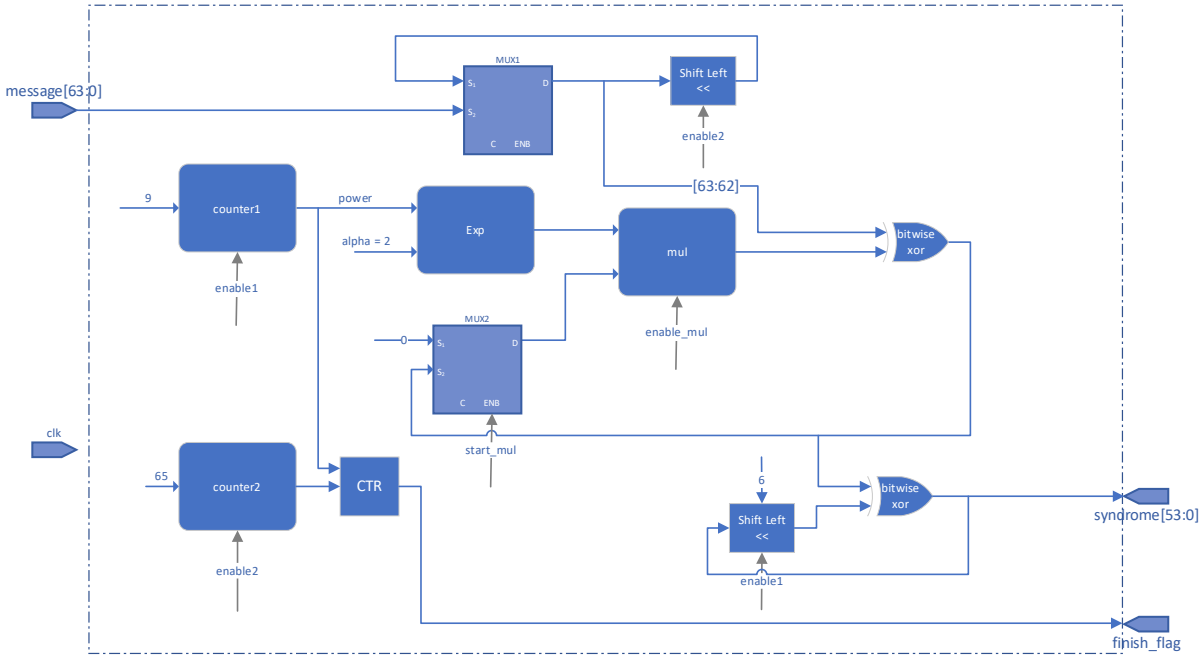


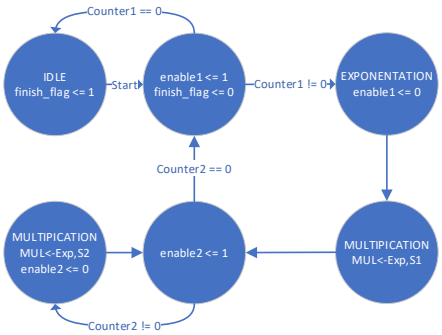


It receives a BCH encoded binary message that has been demodulated from DNA (39 data bits that are potentially damaged + 24 ECC bits). The decoder can fix up to 4 errors that could occur during the DNA synthesis process, the chance for more errors is slim to none[1]. The encoded message passes through the 3 upper modules in a sequence:

Syndrome Calculation: compute the syndromes for the encoded message and if it is zero then finish. Else - continue to Key Equation Solver which computes the root of the error location polynomial g, if it is zero then finish. Else – compute the error locations binary array and perform a XOR operation between it and the encoded message. Finally, return the 39 MSB bits which are the data bits.

5.4.1 Syndrome Calculation





The syndromes are:

$$s_i = message\left(\alpha^i\right) \text{ for } i = 1,2,\ldots,8$$

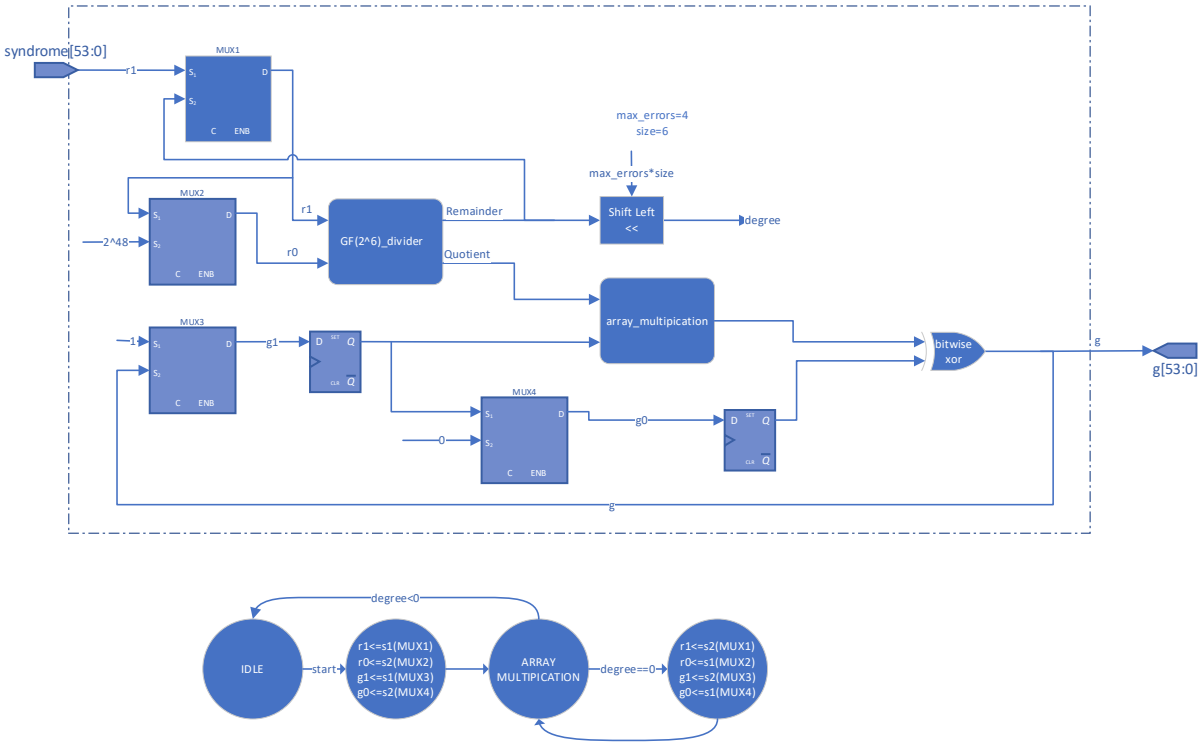
If the syndromes are all zero, then the decoding is done. Otherwise save the non-zero syndromes in an array.

Syndrome calculation process :

For $i = 8$ until $i = 1$:

1. Calculate α^i.
2. $s_i = sum =$
   $\left(\left(\left(\left(\left(msg[63]\right)\alpha^i + msg[62]\right)\alpha^i + msg[61]\right)\alpha^i + \ldots + msg[1]\right)\alpha^i + msg[0]\right)$
3. Reset sum and shift the syndrome array to the left, adding s_i in to it.

5.4.2 Key Equation Solver

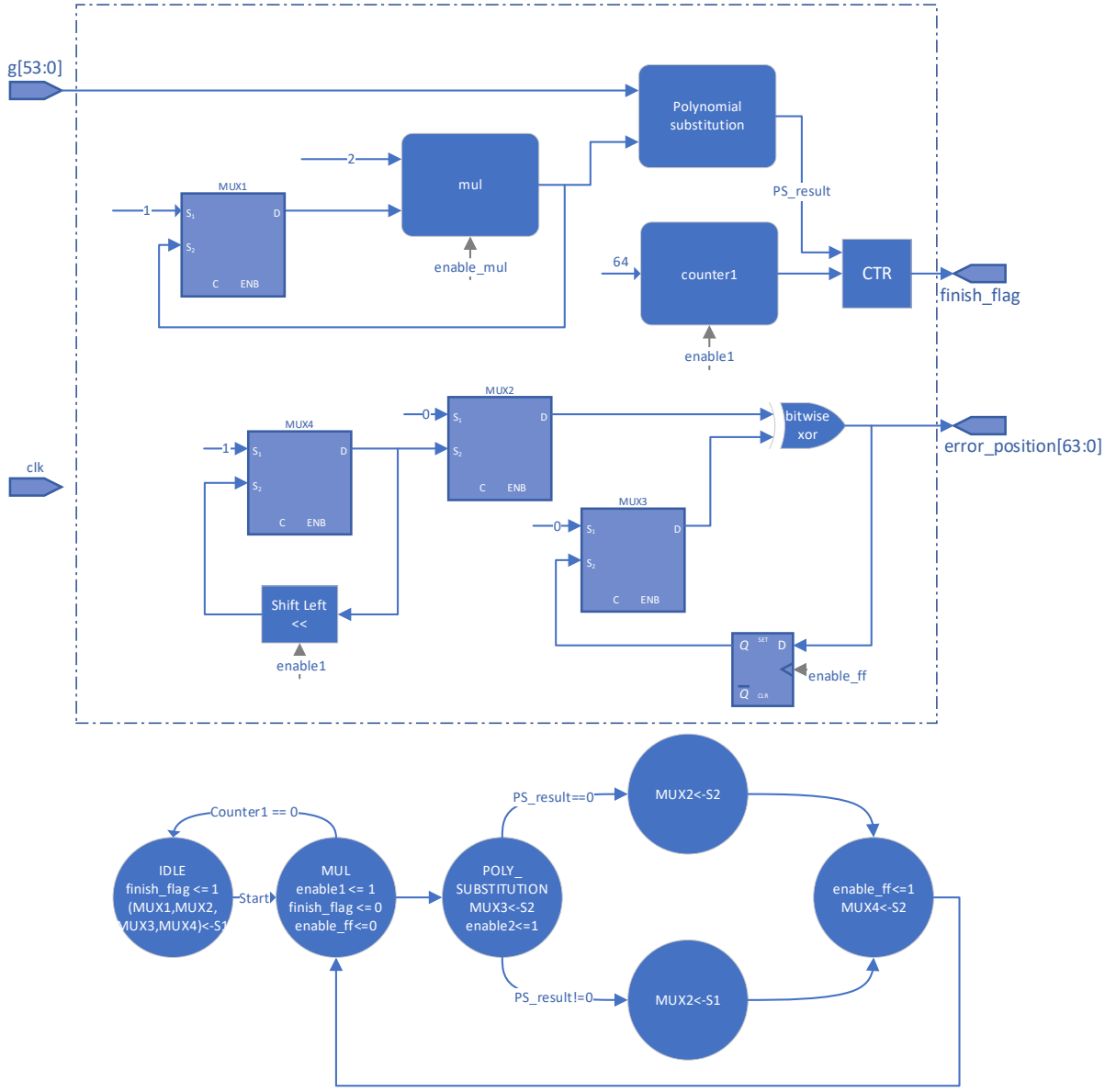


Initialize $r_0 = 2^{48}$, $r_1 = Syndrome$, $g_0 = 0$ and $g_1 = 1$,

Find the error locating polynomial:

1. Use the $GF\left(2^6\right)$ divider to divide $r_0(x)$ by $r_1(x)$ and get the quotient $q(x)$, and the remainder $r(x)$.
2. Update $g(x) = g_0(x) - q(x) \cdot g_1(x)$.
3. Check if the degree of $r(x)$ is less than max_errors (4) if true finish.
4. Update $r_0(x) = r_1(x)$, $r_1(x) = r(x)$, $g_0(x) = g_1(x)$ and $g_1(x) = g(x)$. Repeat step 1.

5.4.3 Error location calculation

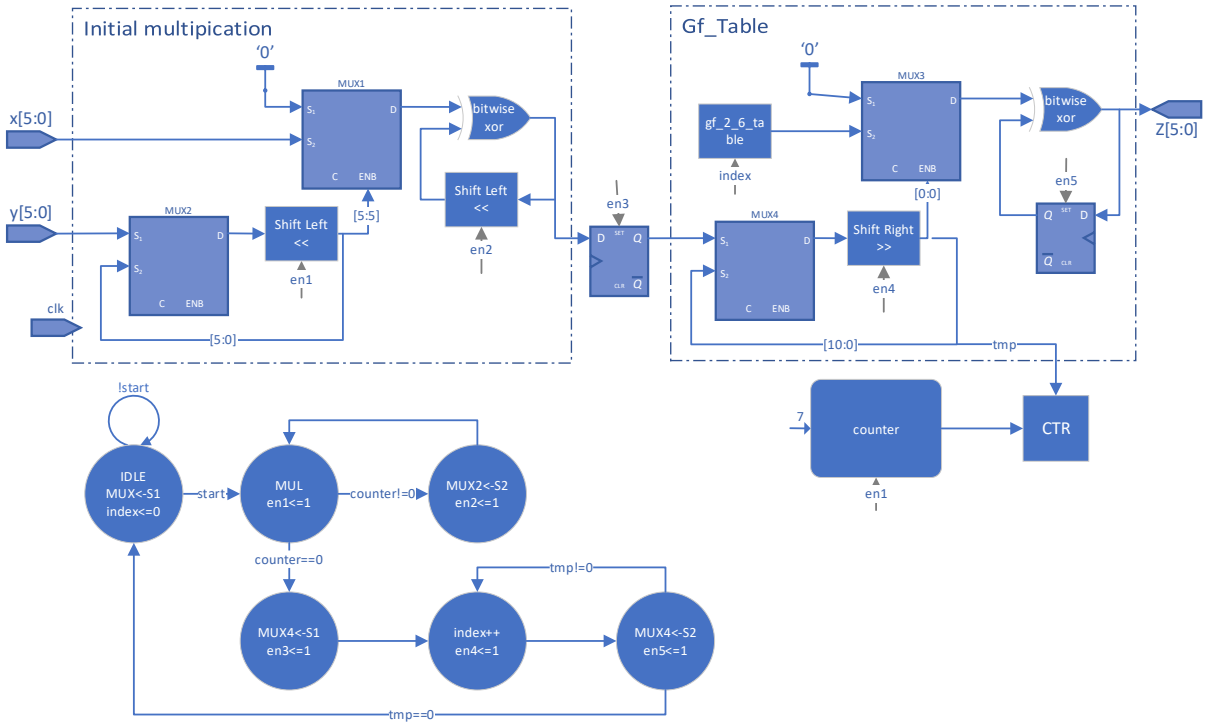


Initialize $error\_position = 0$, $error\_bit[63:0] = 1$. for $1 \le i \le 63$

1. Calculate α^i and then $g(\alpha^i)$ using the polynomial substitution unit from 5.5.6.
2. If $g(\alpha^i) = 0$, then there is an error in position i, turn on the i[th] bit in $error\_position$ by calculating:
   $error\_position = error\_position \oplus error\_bit[63:0]$.
3. Shift $error\_bit[63:0]$ once to the left.

5.5 Galois Field Arithmetic Units

5.5.1 Multiplier

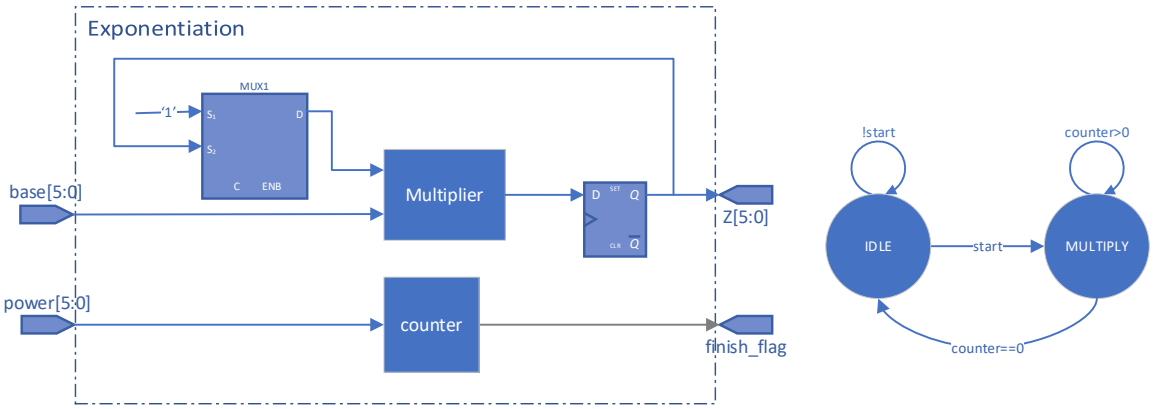


We split the $GF\left(2^6\right)$ multiplication into two stages:

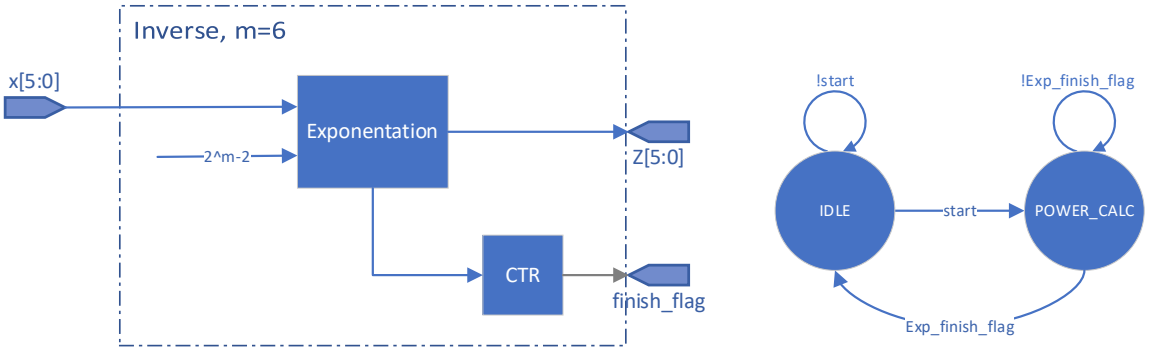
Stage 1: Multiplication between two binary arrays of size 6, the result is a binary array of size 11.

Stage 2: Each index in the binary array represents a number in $GF\left(2^6\right)$, for example (Index, num represented) = {(1,1),(2,2),(3,4),(4,8),(5,16),(6,32),(7,3),(8,6)…}
Return the sum of indexes that contain 1.

5.5.2 Exponentiation

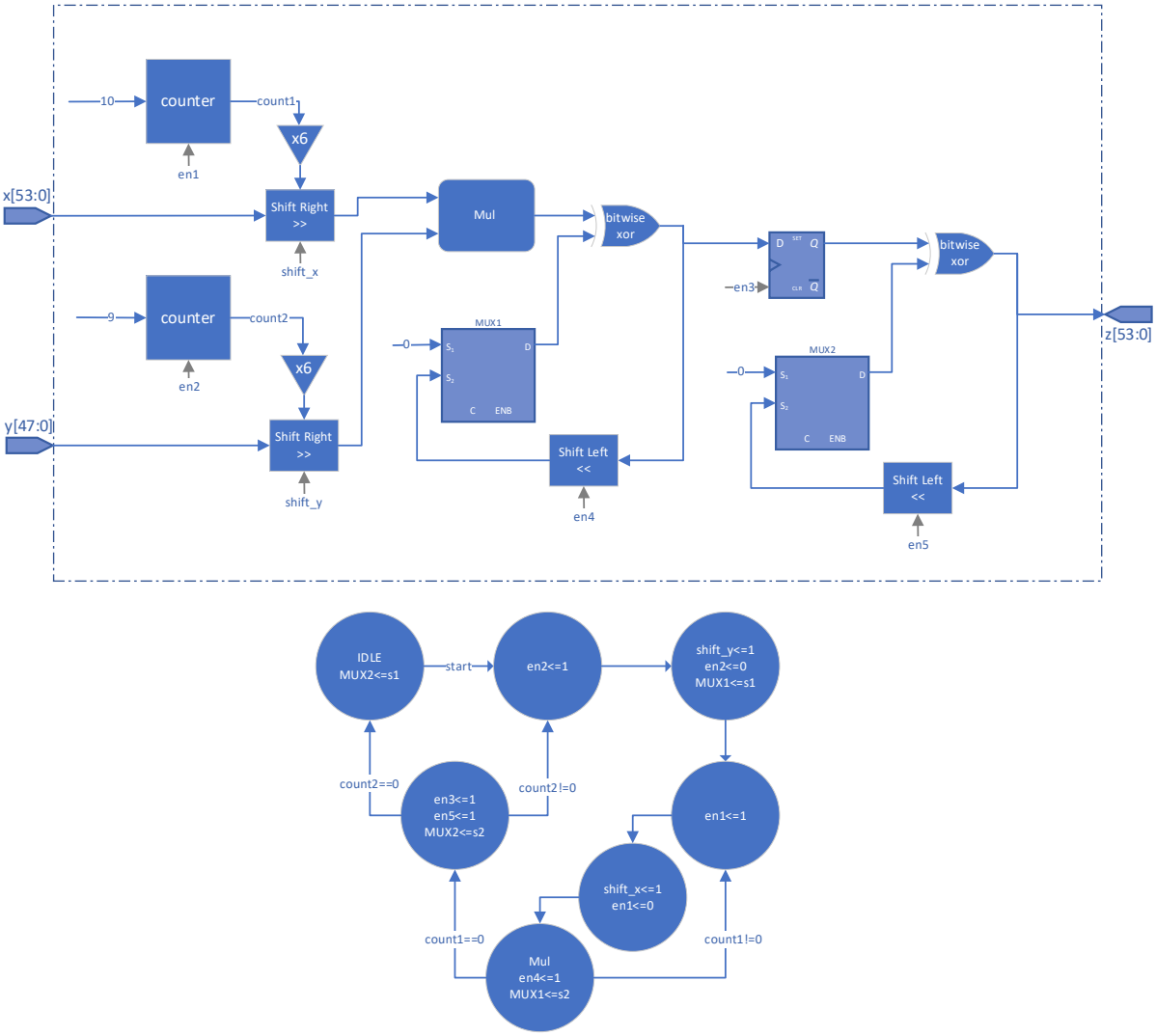


5.5.3 Inverse



In $GF\left(2^m\right)$, $x^{2^m-2} = x^{2^m-1} \cdot x^{-1} = 1 \cdot x^{-1} = x^{-1}$, so to calculate the inverse of x we need to calculate x^{2^m-2}.

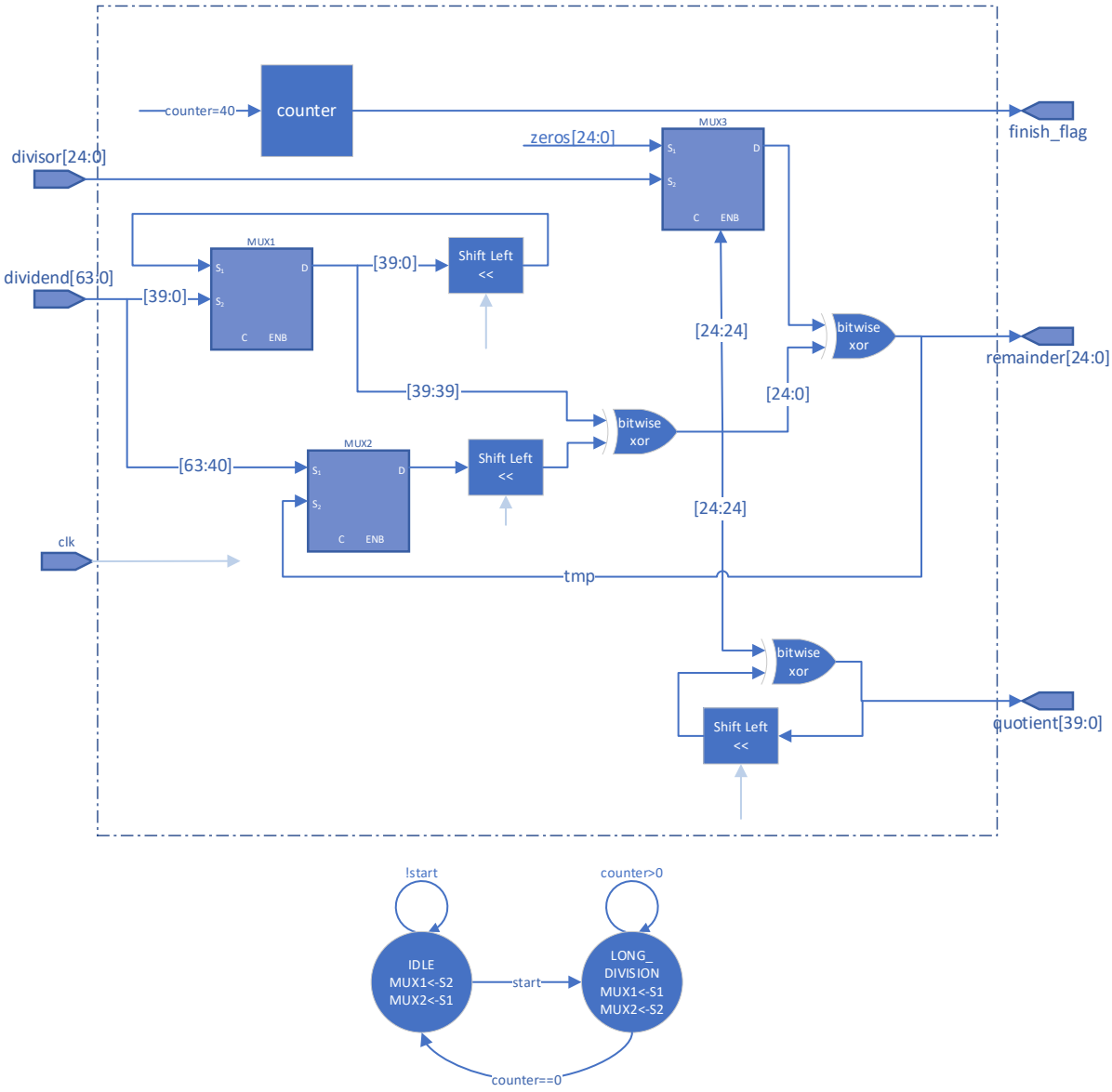
5.5.4 Array Multiplication



We receive 2 arrays: x of size 9 and y of size 8 (each number in the array is 6 bits).

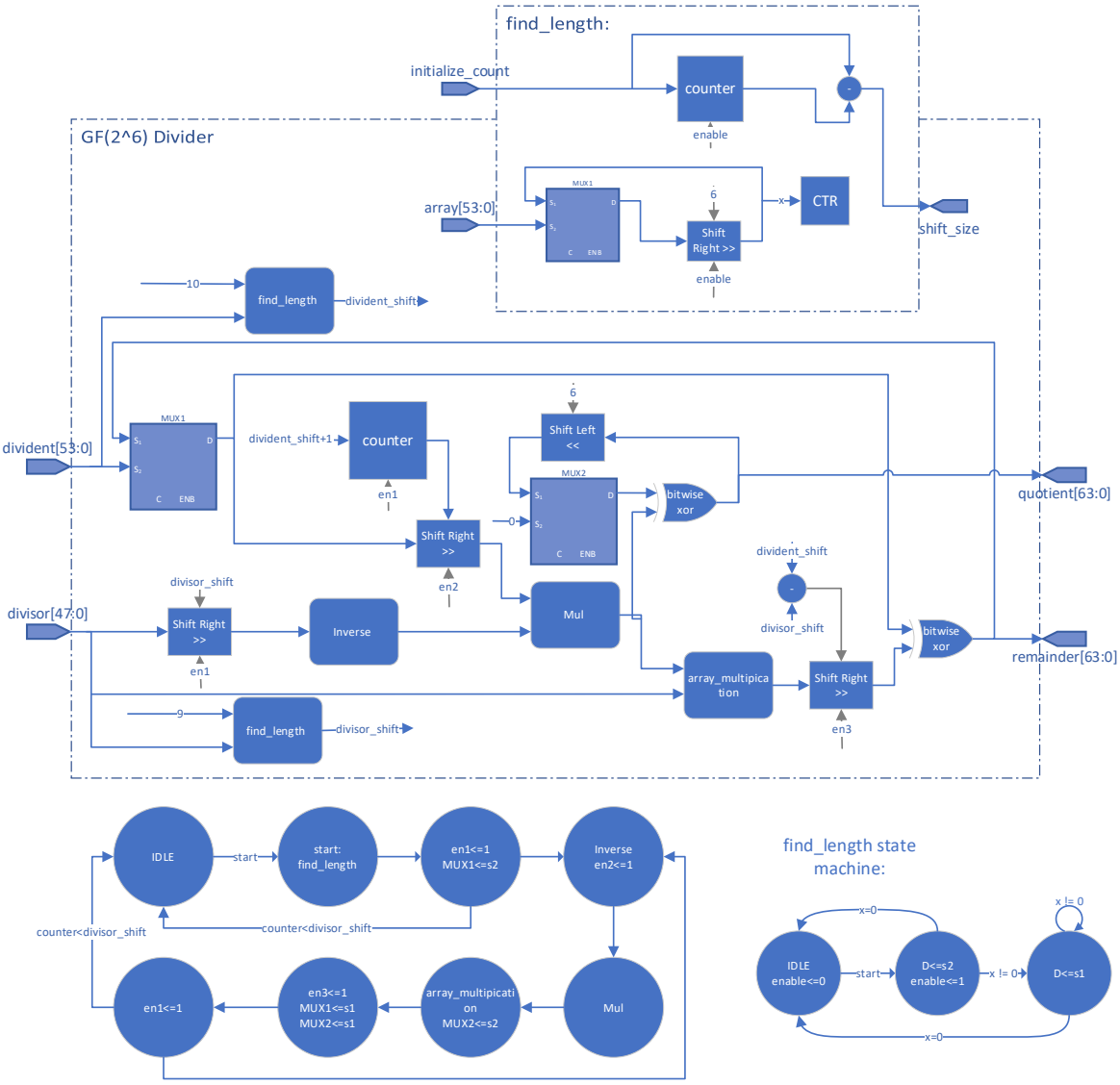
Get last number of y multiply it in $GF\left(2^{6}\right)$ with each number in x (by doing the same, starting with the last number in x and repeating the process until we have multiplied it with every number in x) and then shift the result by the size of the numbers (6) and take the next number in y and repeat the process until we multiplied it with every number in y.

5.5.4 GF(2) Divider



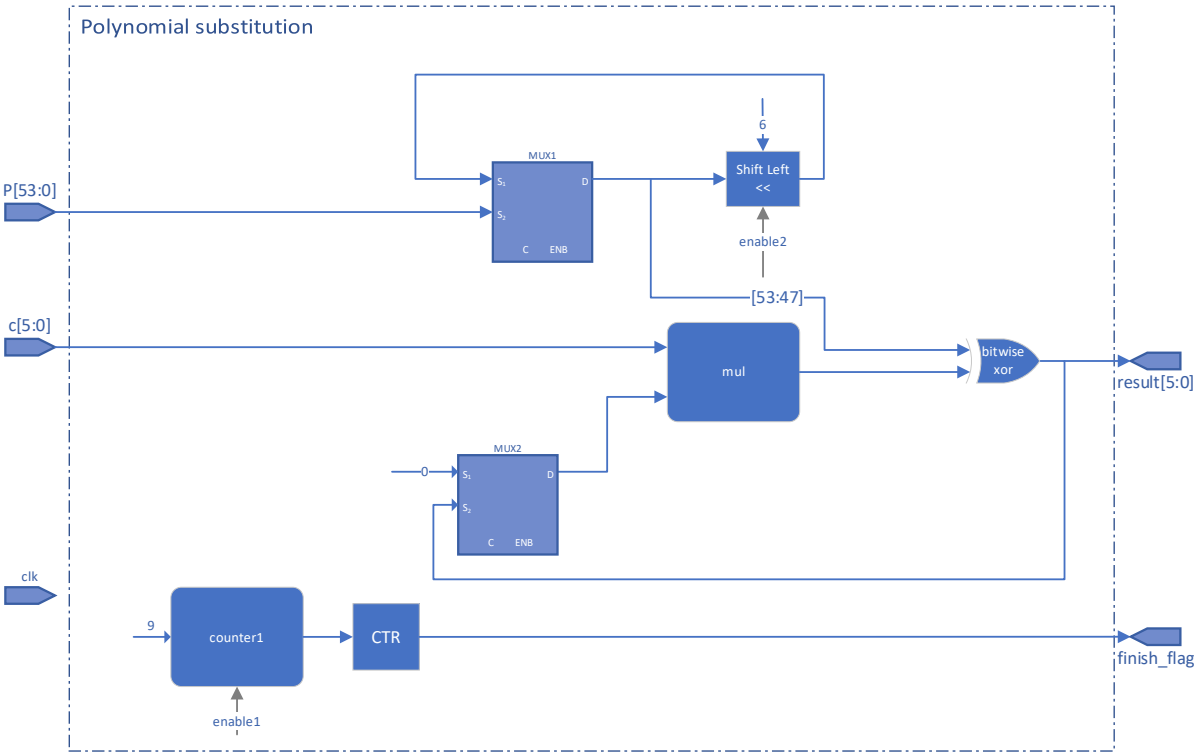
A long division implementation in GF(2). The divisor is 25 bits long, therefore we must copy the 25 MSB bits from the dividend to a variable named tmp (temporary) and then: if the MSB of the tmp is 1, subtract the divisor from the tmp and append 1 to the end of the quotient, otherwise if the MSB is 0, subtract 0 from the tmp and append 0 to the end of the quotient. Either way the MSB will now be 0 in the tmp, we purge it and append the next bit from the dividend to the remaining 24 bits. This process is repeated until the end of the dividend is reached, the remainder of the operation is the tmp.

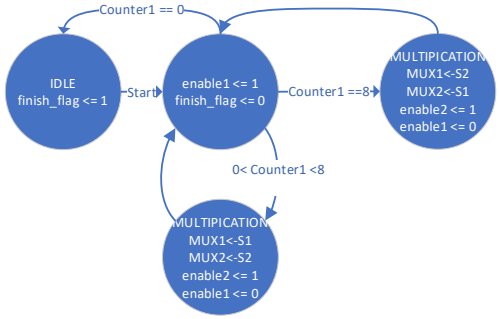
5.5.5 GF(2^6) Divider



1. Find the real length of the divisor (MSB which is not zero).
2. Find the inverse of the MSB.
3. Remainder = dividend.
4. Take MSB of Reminder and multiply it with the inverse number.
5. Insert the result to the quotient.
6. Multiply the result by the divisor.
7. Take [MSB bit -: size of divisor] from the dividend and subtract it from the result in 6.
8. Repeat the process from 4, while the reminder size is bigger than the divisor's real size.

5.5.6 GF(2^6) Polynomial substitution (degree 8)





Receive polynomial $P(x) = a_8x^8 + \ldots + a_1x^1 + a_0$, $c \in GF(2^6)$ and calculate $P(c)$. Input $P[53:0] = (a_8, \ldots, a_1, a_0)$ and c. Output $result = ((((((a_8)c + a_7)c + a_6)c + \ldots + a_1)c + a_0)$.

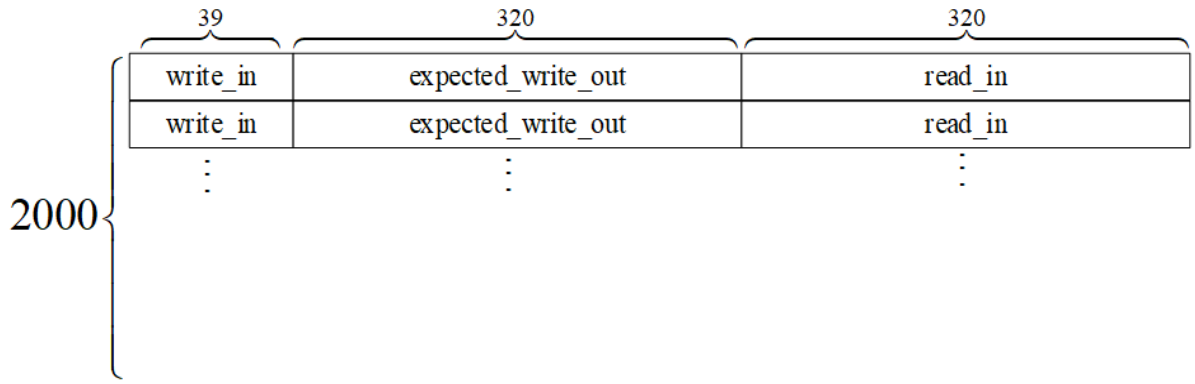
Polynomial substitution algorithm:

1. Initialize $i = 8$, $result = a_8$.
2. Calculate $result \cdot c$ using the multiplier from 5.5.1.
3. $result = result + a_{i-1}$.
4. $i = i - 1$.
5. If $i \neq 0$ repeat the process from step 2. Else return the $result$.

6. Verification

We ran several simulations on 78 kilobits of random data (2000 binary messages of length 39). Using the software simulation from section 4, we generated a test vector, a file with the following structure:

2000 rows, each containing a binary array of size 679.



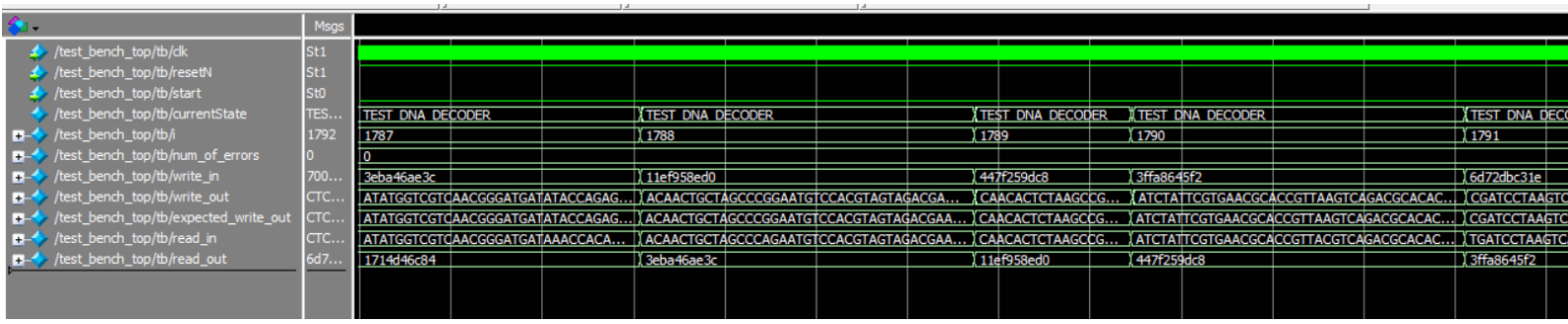
Each binary array contains the following:

- Write_in: The input into the writing block. 39 bits of data that is to be encoded using BCH error correcting code and saved in the DNA as nucleotides.
- Expected_write_out: The output from the write operation in the software simulation. 40 ASCII characters (320 binary bits) representing 40 nucleotides, to be compared against the 40 nucleotides that are outputted from the write block that we implemented.
- Read_in: The DNA string contaminated by randomly swapping up to 2 nucleotides. It is to be translated back into binary and decoded to restore the original data, then compared to write_in to check the validity of the read block.

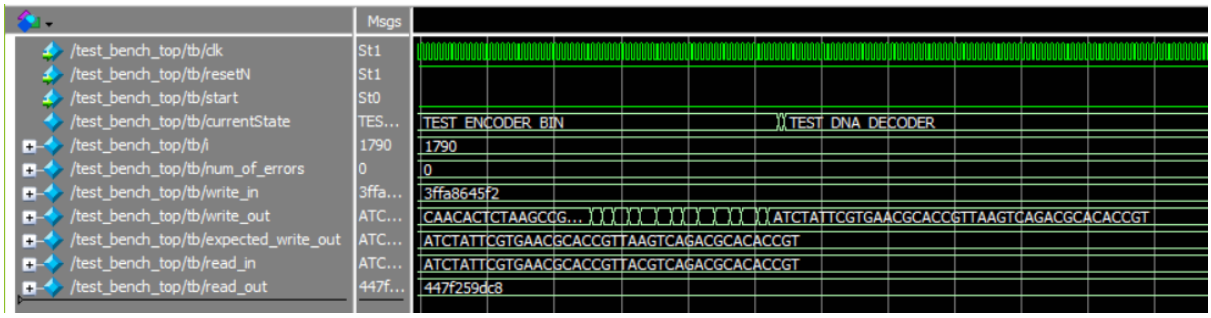
Here is an example of one such simulation,

```
# Compile of bin_to_dna.sv was successful.
# Compile of decoder.sv was successful.
# Compile of dna_to_bin.sv was successful.
# Compile of encoder.sv was successful.
# Compile of error_location.sv was successful.
# Compile of gf_table.sv was successful.
# Compile of gf26_deconv.sv was successful.
# Compile of gf26_exp.sv was successful.
# Compile of gf26_inverse.sv was successful.
# Compile of gf26_mul_array.sv was successful.
# Compile of key_equation_solver.sv was successful.
# Compile of mul.sv was successful.
# Compile of mul_controller.sv was successful.
# Compile of syndrome_calculation.sv was successful.
# Compile of test_bench.sv was successful.
# Compile of test_bench_top.sv was successful.
# Compile of compute_value.sv was successful.
# Compile of gf2_deconv.sv was successful.
# Compile of top.sv was successful.
# 19 compiles, 0 failed with no errors.
```

```
# Loading work.test_bench_top
# Loading work.test_bench_sv_unit
# Loading work.test_bench
# Loading work.top_sv_unit
# Loading work.top
# Loading work.encoder
# Loading work.gf2_deconv
# Loading work.bin_to_dna
# Loading work.dna_to_bin
# Loading work.decoder
# Loading work.syndrome_calculation
# Loading work.mul_controller
# Loading work.mul
# Loading work.gf_table
# Loading work.key_equation_solver
# Loading work.gf26_deconv
# Loading work.gf26_inverse
# Loading work.gf26_exp
# Loading work.gf26_mul_array
# Loading work.error_location
# Loading work.compute_value
VSIM 13> run -all
# Testing complete, total number of errors found:          0
# Number of bits written and read: {      78000}
# ** Note: $stop    : C:/Users/jadht/OneDrive - Technion/winter2020/project/SystemVerilog_files/test_bench_top.sv(41)
#    Time: 164228012 ns  Iteration: 1  Instance: /test_bench_top
```

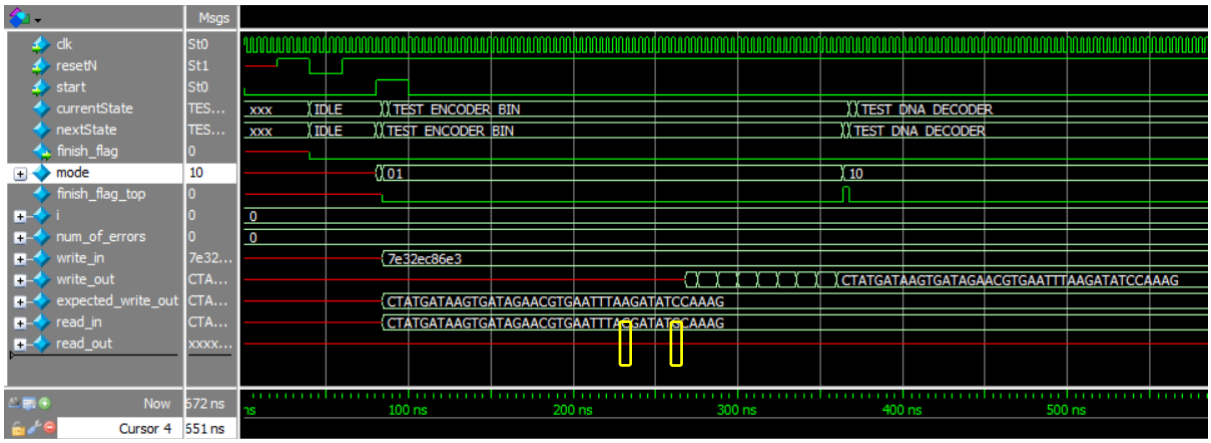


We can see that the output for both the read and write operations matches the software simulation. We test the read operation in the state “TEST_DNA_DECODER” which provides the output in the next cycle, and the write operation in the state “TEST_ ENCODER_BIN”. The write operation cannot be seen in the image above because the decoding operation takes most of the time. It can be seen when zoomed in more:



To show the process that takes a binary message to DNA and back, we ran a simulation on a random binary message.

6.1 Test Bench



The test bench receives an input vector (write_in, expected_write_out, read_in) as explained above. The test bench is separated into two different phases:

Phase one tests the encoder and the mapping from binary to DNA. It takes the write_in array (39 bits), adds ECC bits to it using the encoder which produces 63 bits, and finally maps the 63 bits to 40 ASCII characters using the binary_to_dna unit, these ACII characters are the write_out array which is supposed to be equal to expected_write_out (* condition 1).

Phase two tests the decoder and the mapping from DNA to binary. It takes the read_in array (40 ASCII characters) which is the DNA after the write operation but with a high chance of containing synthesis errors (in this example there are 2 errors, highlighted in yellow in the figure above), it maps it to a 63 bits binary array using the dna_to_binary unit, and finally using the decoder it retrieves the original binary message which is supposed to be equal to write_in(* condition 2).

Signals in this module:

Clk, reset, start, currentState, nextState.

finish_flag - turned on when the testing is finished.

mode 0 - IDLE state do nothing.

1 – start signal to begin the ENCODER_BIN phase.

2 – start signal to begin the DNA_DECODER phase.

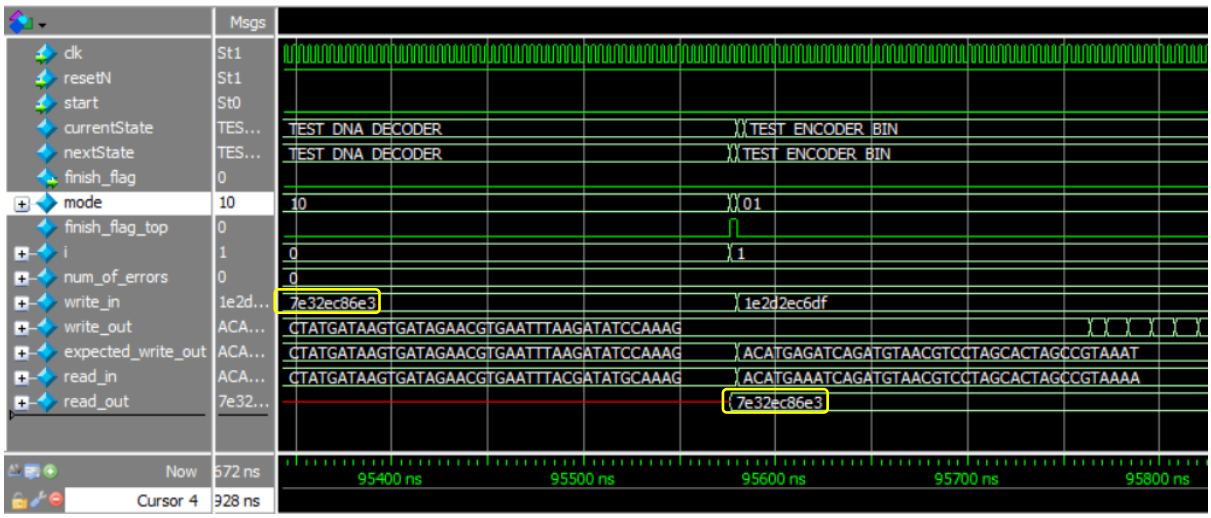
finish_flag_top - turned on when either phase is finished.

i - representing the index of the current test 0-1999 (0 in this sample).

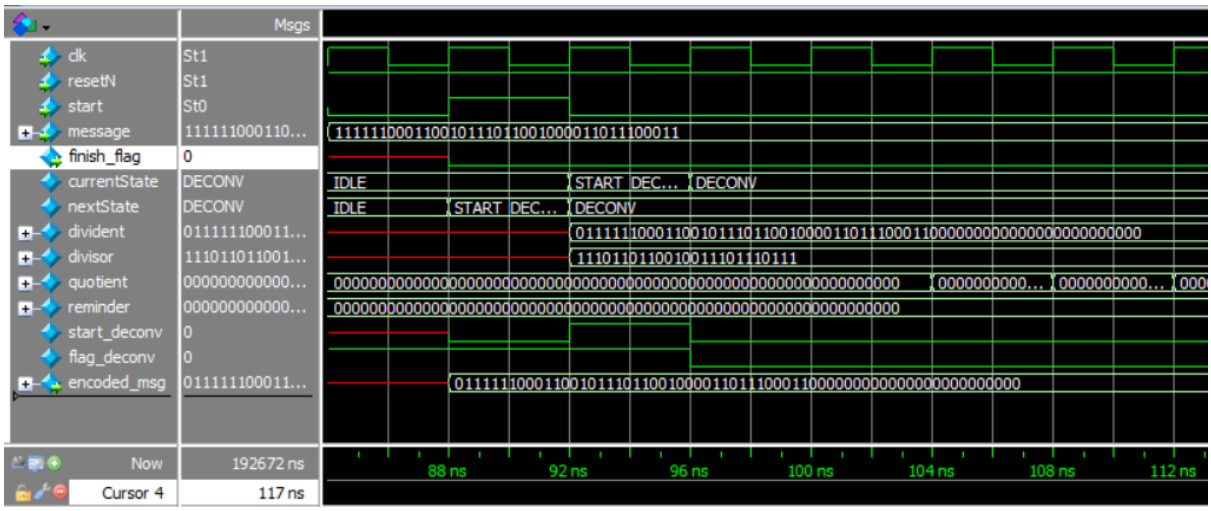
num_of_errors - initialized to zero and increased by 1 whenever conditions 1 or 2 are not met.

write_out- output of phase 1 (ASCII characters representing DNA nucleotides).

read_out- output of phase 2 (binary message).



6.2 Encoder



The encoder receives an input message of 39 binary bits. It shifts the message 25 bits to the left and sends a start signal to activate the deconv unit, which has three output signals (quotient, remainder and flag_deconv). When flag_deconv turns on, the encoder takes the remainder and adds it to the shifted binary message and returns it as the output (named encoded_msg).

Signals in this module:

Clk, reset, start, currentState, nextState.

Message - the binary message to be encoded.

finish_flag - turned on when the encoder is finished.

Dividend – the shifted binary message to be encoded.

Divisor - the polynomial generator constant.

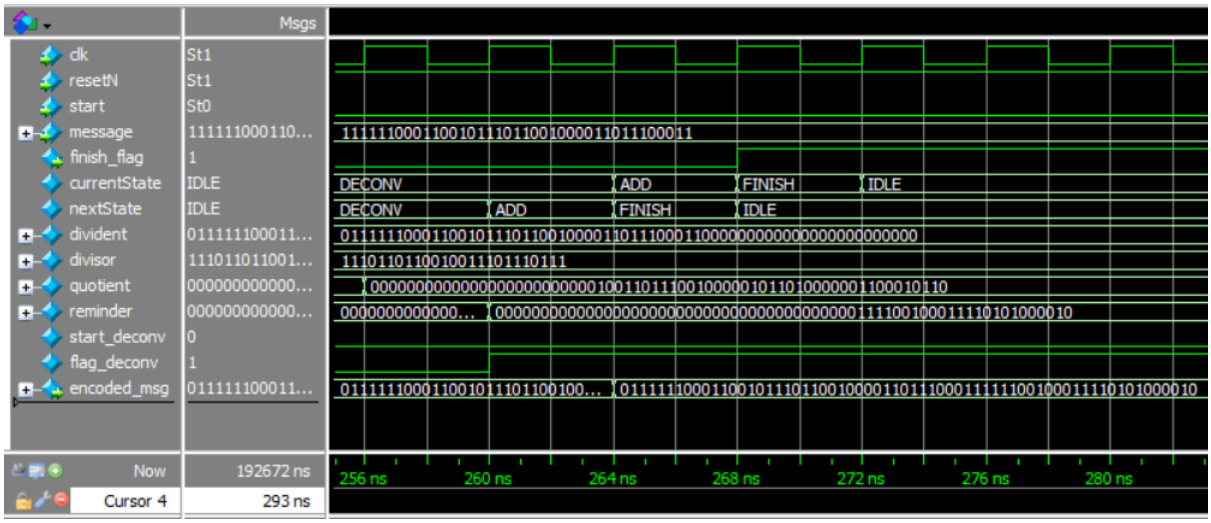
Quotient – the quotient result of the division.

Remainder – the remainder of the division.

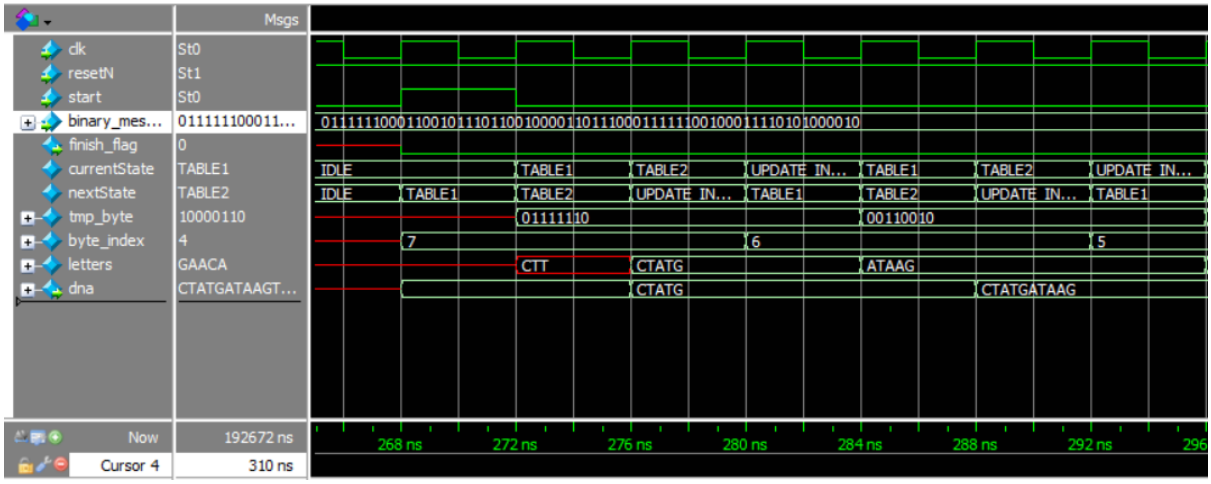
Start_deconv – signal to activate the deconv unit.

flag_deconv - turns on when deconv finishes.

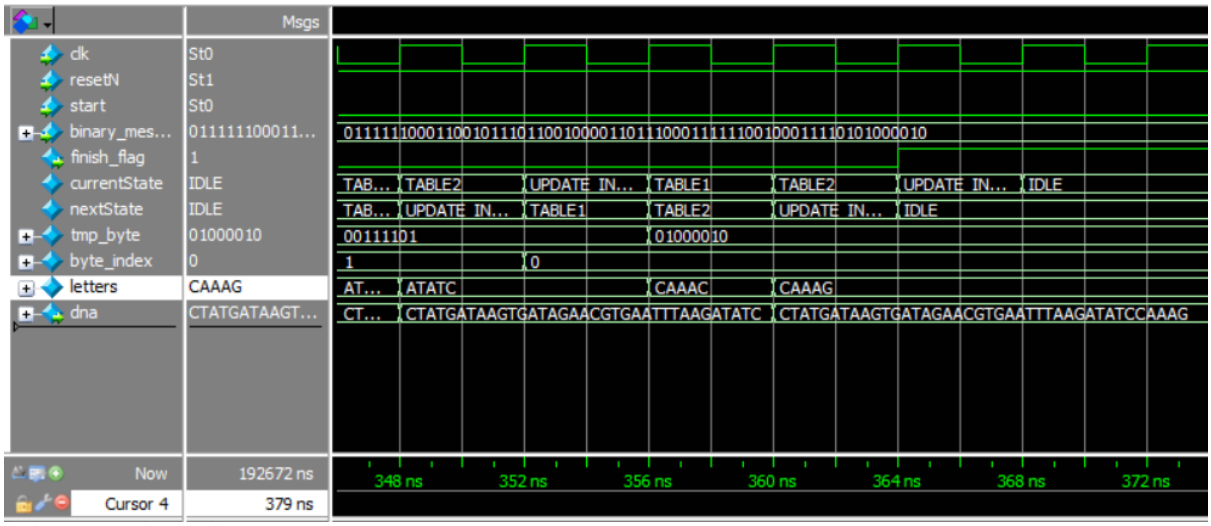
Encoded_message – the final output of the encoder.



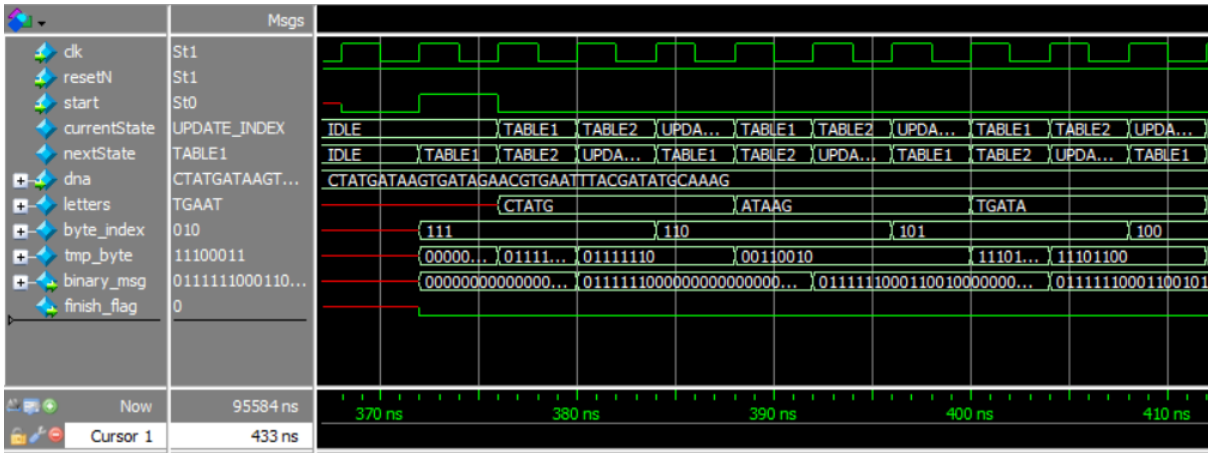
6.3 Binary to DNA



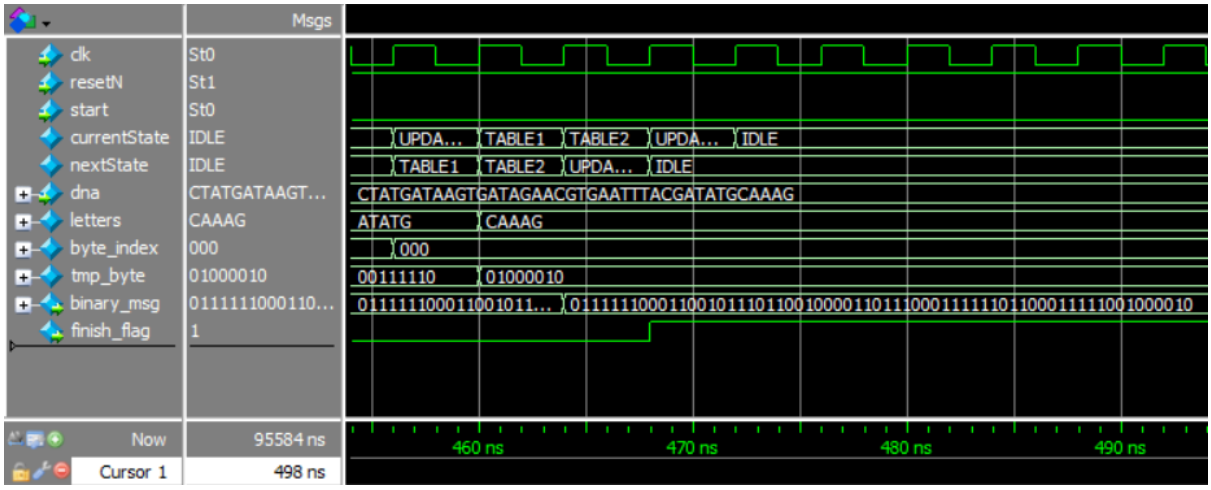
The binary to DNA module receives the encoded message and splits it into segments of 8 bits. It then maps each segment into 5 nucleotides using tables 1 and 2. Finally it combines all the nucleotides into a DNA string of size 40.



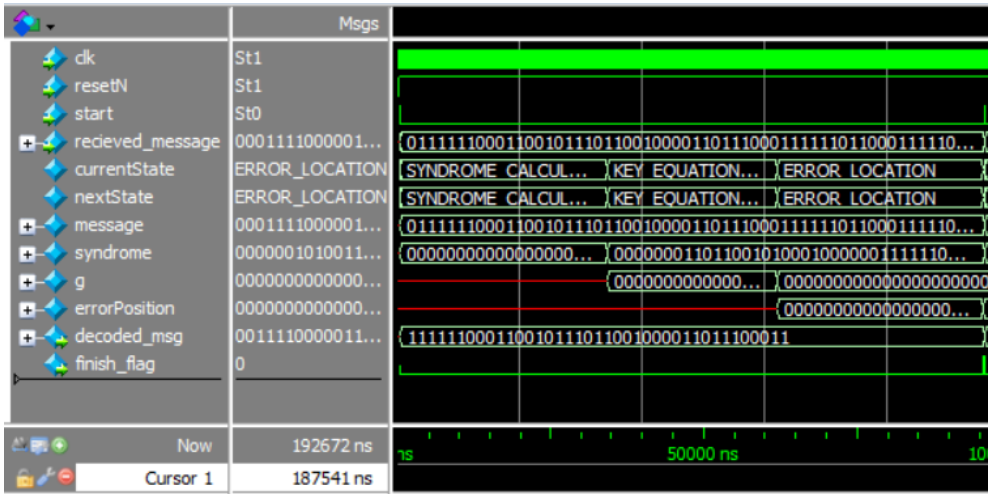
6.4 DNA to Binary



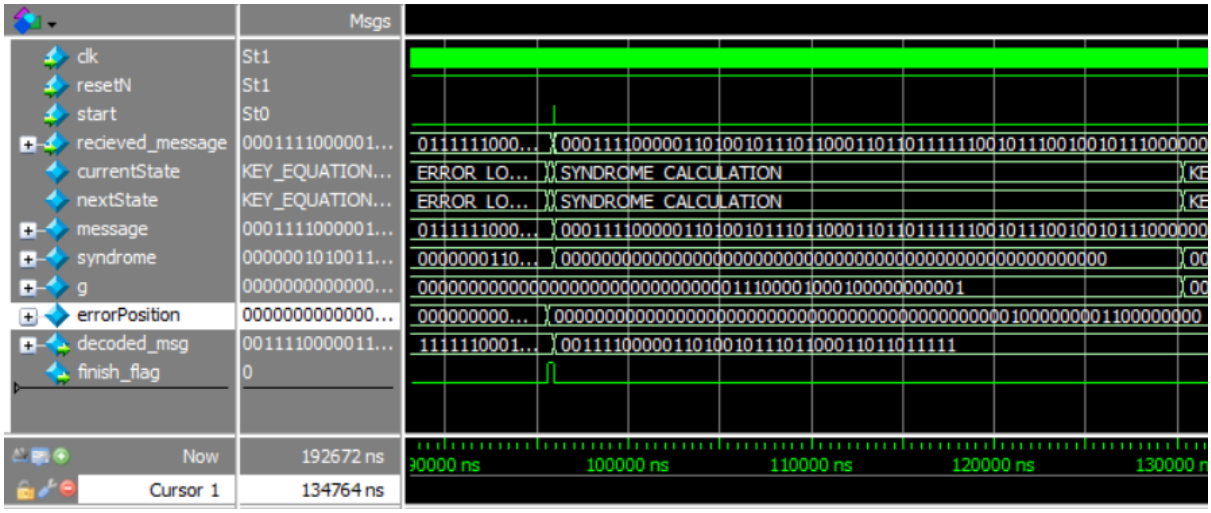
The DNA to binary module receives a DNA string of size 40 and splits it into segments of 5 nucleotides. It then maps each segment into 8 bits using tables 1 and 2. Finally it combines all the bytes into a binary array of size 64.



6.5 Decoder



The decoder receives a binary array of size 64 and passes it through the 3 modules: Syndrome calculation that computes the syndrome, followed by Key equation solver that computes the root of the error location polynomial (g) and finally error location that outputs a mask to fix the errors if any are found. Once the mask is applied, we get the corrected binary message.



7. Synthesis

7.1 Power

```
****************************************
Report : power
        -analysis_effort low
Design : DNA_Controller
Version: N-2017.09-SP3
Date   : Fri ApAr 30 16:22:52 2021
****************************************


Library(s) Used:

tsl18fs120_typ(File:/tools/kits/tower/PDK_TS18SL/FS120_STD_Cells_0_18um_2005_12/DW_TOWER_tsl18fs120/2005.12/synopsys/2004.12/models/tsl18fs120_typ.db)


Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ
Wire Load Model Mode: enclosed

Design        Wire Load Model            Library
------------------------------------------------
DNA_Controller         35000             tsl18fs120_typ
encoder                4000              tsl18fs120_typ
bin_to_dna             4000              tsl18fs120_typ
dna_to_bin             4000              tsl18fs120_typ
decoder                35000             tsl18fs120_typ
gf2_deconv             4000              tsl18fs120_typ
syndrome_calculation   4000              tsl18fs120_typ
key_equation_solver    35000             tsl18fs120_typ
error_location         8000              tsl18fs120_typ
mul_controller_0       4000              tsl18fs120_typ
gf26_deconv            16000             tsl18fs120_typ
gf26_mul_array_0       4000              tsl18fs120_typ
compute_value          4000              tsl18fs120_typ
mul_0                  4000              tsl18fs120_typ
gf_table_0             4000              tsl18fs120_typ
gf26_inverse           4000              tsl18fs120_typ
gf26_exp               4000              tsl18fs120_typ
mul_controller_1       4000              tsl18fs120_typ
mul_controller_2       4000              tsl18fs120_typ
mul_controller_3       4000              tsl18fs120_typ
mul_controller_4       4000              tsl18fs120_typ
mul_controller_5       4000              tsl18fs120_typ
mul_controller_6       4000              tsl18fs120_typ
gf26_mul_array_1       4000              tsl18fs120_typ
mul_1                  4000              tsl18fs120_typ
mul_2                  4000              tsl18fs120_typ
mul_3                  4000              tsl18fs120_typ
mul_4                  4000              tsl18fs120_typ
mul_5                  4000              tsl18fs120_typ
mul_6                  4000              tsl18fs120_typ
gf_table_1             4000              tsl18fs120_typ
gf_table_2             4000              tsl18fs120_typ
gf_table_3             4000              tsl18fs120_typ
gf_table_4             4000              tsl18fs120_typ
gf_table_5             4000              tsl18fs120_typ
gf_table_6             4000              tsl18fs120_typ
gf2_deconv_DW01_inc_0  ForQA             tsl18fs120_typ
gf26_deconv_DW01_dec_3 4000              tsl18fs120_typ
gf26_deconv_DW01_inc_2 4000              tsl18fs120_typ


Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 117.1590 mW   (95%)
  Net Switching Power  =   5.5938 mW    (5%)
                         ---------
Total Dynamic Power    = 122.7528 mW  (100%)

Cell Leakage Power     = 747.5180 nW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000            0.0000             0.0000  (   0.00%)
register         116.3371           0.1306        4.4053e+05           116.4681  (  94.88%)
sequential         0.0000           0.0000            0.0000             0.0000  (   0.00%)
combinational      0.8229           5.4633        3.0699e+05             6.2864  (   5.12%)
--------------------------------------------------------------------------------------------------
Total            117.1599 mW        5.5939 mW     7.4752e+05 pW       122.7545 mW
```

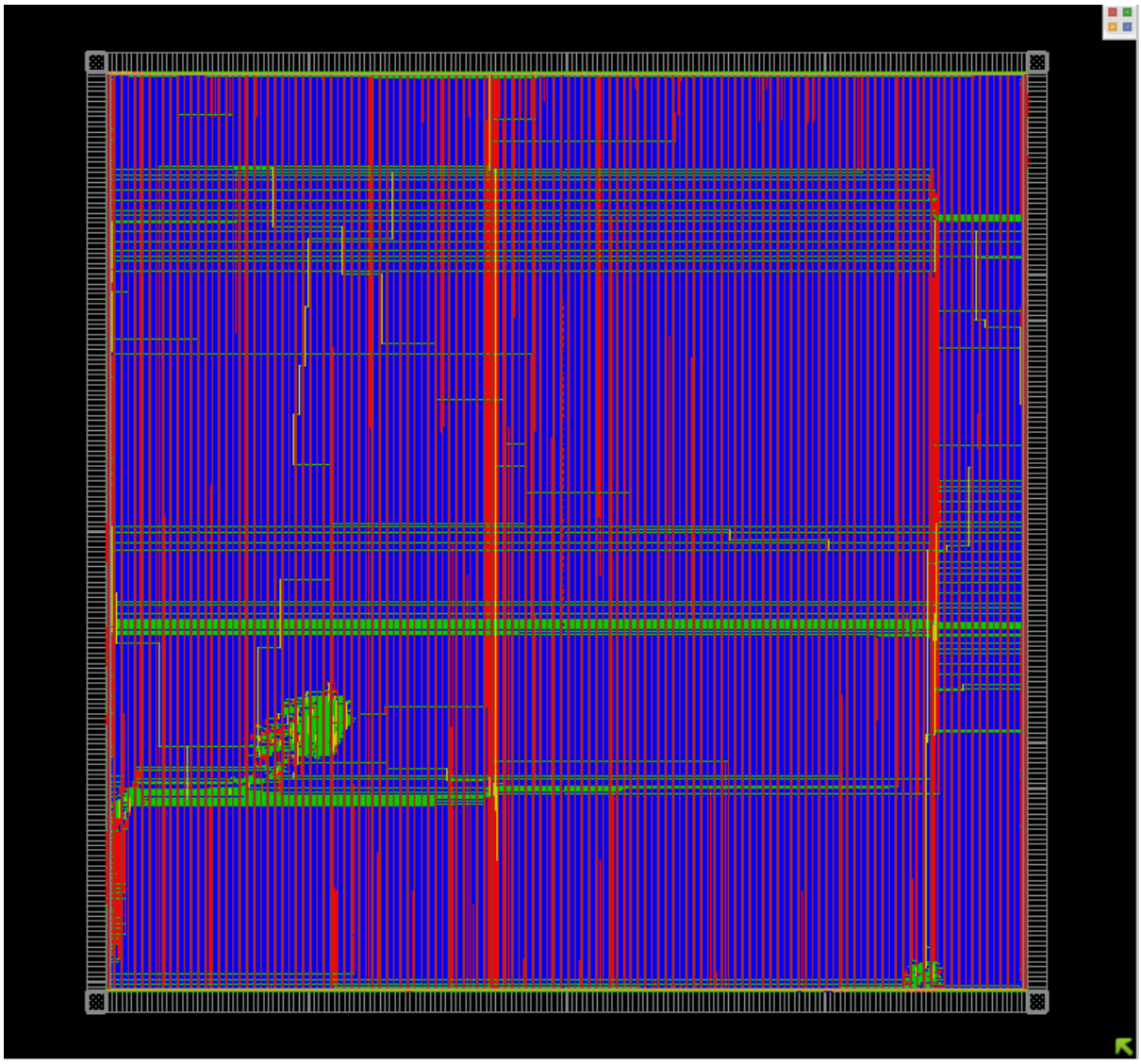
7.2 Area

```
****************************************
Report : area
Design : DNA_Controller
Version: N-2017.09-SP3
Date   : Fri Apr 30 16:22:24 2021
****************************************

Library(s) Used:
tsl18fs120_typ(File:/tools/kits/tower/PDK_TS18SL/FS120_STD_Cells_0_18um_2005_12/DW_TOWER_tsl18fs120/2005.12/synopsys/2004.12/models/tsl18fs120_typ.db)

Number of ports:                        3586
Number of nets:                        16747
Number of cells:                       13801
Number of combinational cells:         10257
Number of sequential cells:             3506
Number of macros/black boxes:              0
Number of buf/inv:                      3510
Number of references:                     30

Combinational area:             14388.750000
Buf/Inv area:                    2934.750000
Noncombinational area:          18999.250000
Macro/Black Box area:               0.000000
Net Interconnect area:          11799.562866

Total cell area:                33388.000000
Total area:                     45187.562866
```

7.3 Timing

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
        -sort_by group
Design : DNA_Controller
Version: N-2017.09-SP3
Date   : Fri Apr 30 16:35:47 2021
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ
Wire Load Model Mode: enclosed

  Startpoint: dna2bin/byte_index_out_reg[2]
              (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: dna2bin/tmp_msg_out_reg[8]
            (rising edge-triggered flip-flop clocked by CLK)
  Path Group: CLK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  DNA_Controller     35000                 tsl18fs120_typ
  dna_to_bin         4000                  tsl18fs120_typ

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock CLK (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  dna2bin/byte_index_out_reg[2]/CP (dfcrq1)               0.00 #     0.00 r
  dna2bin/byte_index_out_reg[2]/Q (dfcrq1)                0.35       0.35 f
  dna2bin/U103/Z (or03d1)                                 0.23       0.58 f
  dna2bin/U380/Z (buffd1)                                 0.18       0.77 f
  dna2bin/U49/ZN (inv0d2)                                 0.30       1.07 r
  dna2bin/U329/ZN (aoi22d1)                               0.11       1.17 f
  dna2bin/U57/Z (an04d1)                                  0.22       1.39 f
  dna2bin/U95/ZN (nr02d2)                                 0.11       1.50 r
  dna2bin/U97/Z (an03d0)                                  0.39       1.89 r
  dna2bin/U292/Z (or03d0)                                 0.15       2.04 r
  dna2bin/U291/ZN (oai31d1)                               0.07       2.11 f
  dna2bin/U290/ZN (aoi222d1)                              0.27       2.37 r
  dna2bin/U289/ZN (aoi311d1)                              0.10       2.47 f
  dna2bin/U417/ZN (aon211d1)                              0.17       2.64 r
  dna2bin/U66/Z (buffd7)                                  0.14       2.78 r
  dna2bin/U286/ZN (oaim22d1)                              0.06       2.83 f
  dna2bin/tmp_msg_out_reg[8]/D (dfnrq1)                   0.00       2.83 f
  data arrival time                                                  2.83

  clock CLK (rise edge)                                   3.00       3.00
  clock network delay (ideal)                             0.00       3.00
  dna2bin/tmp_msg_out_reg[8]/CP (dfnrq1)                  0.00       3.00 r
  library setup time                                     -0.17       2.83
  data required time                                                 2.83
  --------------------------------------------------------------------------
  data required time                                                 2.83
  data arrival time                                                 -2.83
  --------------------------------------------------------------------------
  slack (MET)                                                        0.00
```

8. Layout



References

[1] Meinolf Blawat, Klaus Gaedke, Ingo Hütter, Xiao-Ming Chen, Brian Turczyk, Samuel Inverso, Benjamin W. Pruitt, George M. Church: Forward error correction for DNA data storage. Procedia Comput. Sci. 80, 1011–1022. (2016).

[2] Data Storage Supply and Demand Worldwide, from 2009 to 2020 (in exabytes). Statista. Retrievable at: https://www. statista.com/statistics/751749/worldwidedata-storage-capacity-and-demand/

[3] Patrizio, A. ,IDC: Expect 175 zettabytes of data worldwide by 2025, Network World, https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-dataworldwide-by-2025.html. (2018).

[4] Zhirnov, V., Zadegan, R. M., Sandhu, G. S., Church, G. M. & Hughes, W. L. Nucleic acid memory. Nat. Mater. 15, 366–370 (2016).

[5] Wu J, Cunanan J, Kim L, Kulatunga T, Huang C, Anekella B: Stability of Genomic DNA at Various Storage Conditions. SeraCare Life Sciences, Milford, MA. International Society for Biological and Environmental Repositories (ISBER) 2009 Annual Meeting (May 12-15, 2009).

[6] Ceze, L., Nivala, J. & Strauss, K. Molecular digital data storage using DNA. Nat. Rev. Genet. 20, 456–466 (2019).

[7] Dormehl, L. When we run out of room for data, scientists want to store it in DNA. Digital Trends. Retrievable at: https://www.digitaltrends.com/ cool-tech/dna-data-catalog-startup/. (July 8, 2018).

[8] Bishop, B; McCorkle, N; Zhirnov, V. Technology Working Group Meeting on future DNA synthesis technologies. Retrievable at: https://www.src. org/program/grc/semisynbio/semisynbioconsortium-roadmap/6043-full-reportdna-based-storage-final-twg1-4-18.pdf. (September 14, 2017).

[9] The Future of DNA Data Storage. Potomac Institute for Policy Studies. (September 2018).

[10] James Bornholt, Randolph Lopez, Douglas M. Carmean, Luis Ceze, Georg Seelig, Karin Strauss: Toward a DNA-based archival storage system. Published by the IEEE Computer Society. 0272-1732/17/ (May/June 2017).

[11] George M. Church, Yuan Gao, Sriram Kosuri. Next-Generation Digital Information Storage in DNA Science Vol. 337, 28. (September 2012)

[12] Cornish, C. How DNA could store all the world's data in a semitrailer. Financial Times. Retrievable at: https://www.ft.com/content/45ea22b0- cec2-11e7-947e-f1ea5435bcc7 (February 5, 2018).

[13] Danny Dubé, Wentu Song, Kui Cai: DNA Codes with Run-Length Limitation and Knuth-Like Balancing of the GC Contents. The 42nd Symposium on Information Theory and its Applications (SITA2019) Kirishima, Kagoshima, Japan. (Nov. 26–29, 2019).

[14] Todd K. Moon: Error Correction Coding: Mathematical Methods and Algorithms Wiley-Interscience, First Edition (2005).