

Matrix Code Based Multiple Error Correction Technique for N-Bit Memory Data

Done by: SUPERVISED BY: DR. AWS KANAN.

AbdullA AL-MOMANI - 20180776.

kHALID aBU-RBAIA’ - 20190833.

2022

**Table of contents**

Contents

[Abstract 2](#_Toc104775998)

[Introduction 3](#_Toc104775999)

[Proposed Design 4](#_Toc104776000)

[First stage 4](#_Toc104776001)

[First register 4](#_Toc104776002)

[Second stage 5](#_Toc104776003)

[Second register 5](#_Toc104776004)

[Third stage 5](#_Toc104776005)

[Third register 6](#_Toc104776006)

[Fourth Stage 6](#_Toc104776007)

[Fourth register 7](#_Toc104776008)

[Fifth stage 8](#_Toc104776009)

[Fifth register 8](#_Toc104776010)

[Sixth stage 9](#_Toc104776011)

[Sixth register 9](#_Toc104776012)

[Seventh stage 10](#_Toc104776013)

[Design block diagram 11](#_Toc104776014)

[Design Analysis 12](#_Toc104776015)

[Test runs and discussion 12](#_Toc104776016)

[Performance analysis and comparison with non-pipelined architecture 12](#_Toc104776017)

[Conclusion and future work 12](#_Toc104776018)

# Abstract

The hardware design proposed in this report is designed to examine a 32-bit data input to determine whether it has been transmitted correctly, and to correct it to its original state if needed.

The Verilog code used to implement and test this hardware design displays how this accelerator works and shows the results of each stage of this pipeline.

# Introduction

In this project, we will be taking our knowledge of pipelining from the Architecture 1 course, and applying it with our newly found knowledge from the current Architecture 2 course, in order to build a special purpose CPU, which is effectively a matrix-based error detection and correction technique for data in our case.

We will be building a matrix-based error detection and correction pipelined accelerator that is capable of detecting multiple errors, which enhances the performance and throughput for error detection and correction purposes.

We’ve decided to implement this special purpose CPU because it is essential to ensure that our data is preserved and transmitted correctly at all times. This report shows the benefits of having this design implementation being pipelined rather than a single cycle.

The accelerator is applicable in every device that relies on transmitting data either from one component to another, and to enable reliable delivery of digital data over unreliable communication channels.

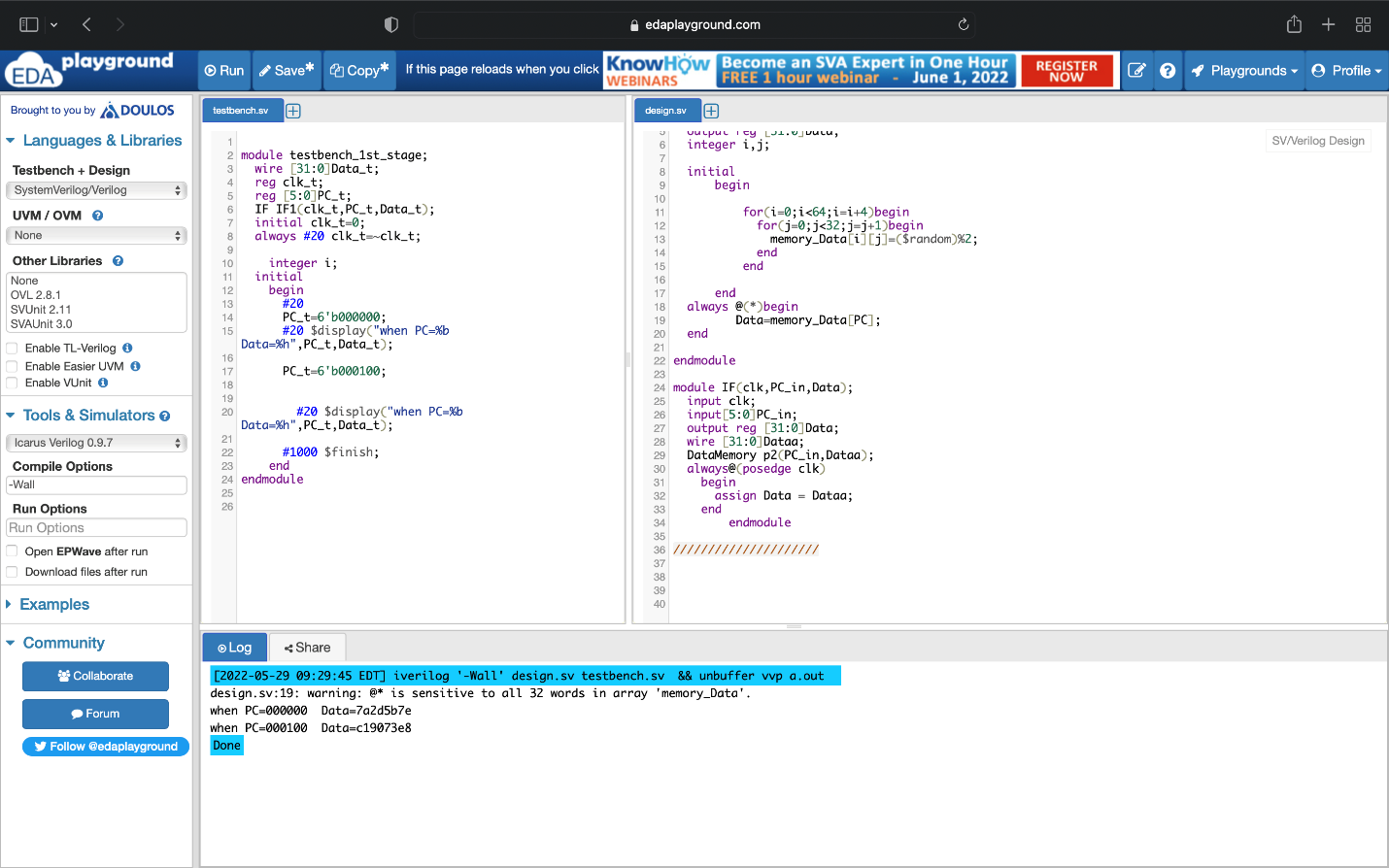
# Proposed Design

Our proposed implemented hardware model consists of 7 stages and 6 registers (buffers).

## First stage

Referred to as IF\_and\_Datamem.

This stage is very straight forward, it fetches the instruction and the data that we’re testing from the memory.



### First register

Receives the data from the previous stage, this register is needed as we’ll be forwarding this data as it is in upcoming stages.

## Second stage

Referred to as the Encoder.

1. Enter the data to encoder that generates the check bits and parity bits.
2. Generate parity bits by using the following formula:

PL = XL ⊕ XL+8 ⊕ XL+16 ⊕ XL+24 where L is the column number from 0 to 7 for eight parity bits.

1. Generate check bits by CNEW=Cj+(cb\*r) and XNEW=Xi+(k2\*r) where cb is the number of check bits per row, r is the row number from 0 to 3, j is the corresponding check bit’s position in the first row and i is the corresponding data bit’s position in the first row.

C[0] = X[0] ⊕ X[1] ⊕ X[3] ⊕ X[4] ⊕ X[6].

C[1] = X[0] ⊕ X[2] ⊕ X[3] ⊕ X[5]⊕ X[6].

C[2] = X[1] ⊕ X[2] ⊕ X[3] ⊕ X[7].

C[3] = X[4] ⊕ X[5]⊕ X[6] ⊕ X[7].

C[4] = X[0] ⊕ X[1] ⊕ X[2] ⊕ X[3] ⊕ X[4] ⊕ X[5]⊕ X[6] ⊕ X[7].

1. Send them to the next register.



### Second register

Receives the check bits and parity bits from the previous stage.

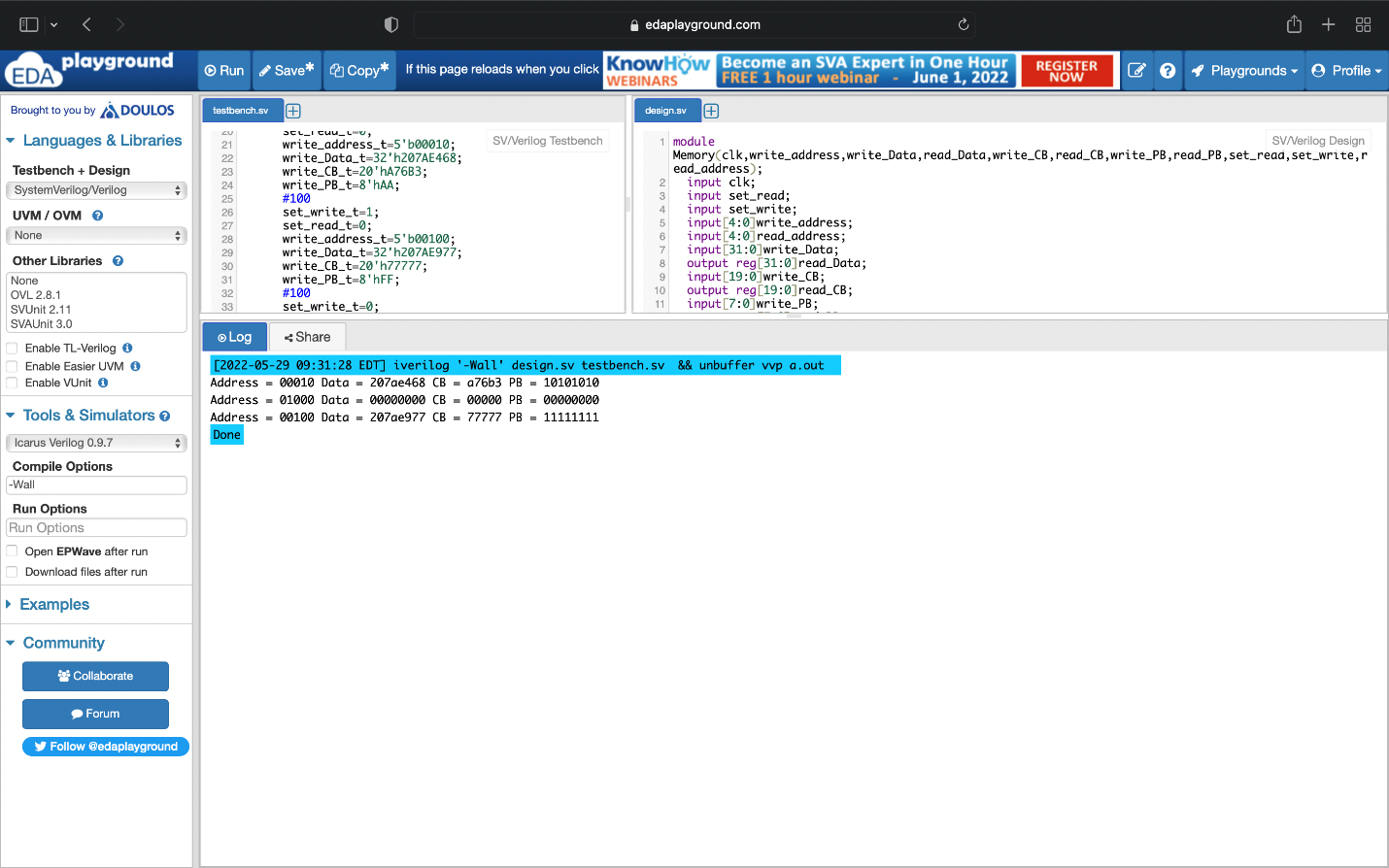
## Third stage

Referred to as Memory.

1. Store the data (32 bits), check bits (20 bits), and parity bits (8 bits) in the memory (64 bits for each index with the last 4 bits being unused).

The memory has read\_address and write\_address signals, we set the value to 1 to choose the appropriate signal.

1. Get the address of the data, check bits and parity bits.
2. Send them to next register.



### Third register

Receives the forwarded data, check bits, parity bits, the addresses of the data, check bits, and parity bits from the previous stage.

## Fourth Stage

Referred to as the prime generator.

1. Enter the data, check bits and parity bits to the prime generator.
2. Use the data that we get from memory to generate parity and check bits prime.

For check bits prime:

C'[0] = X[0] ⊕ X[1] ⊕ X[3] ⊕ X[4] ⊕ X[6].

C'[1] = X[0] ⊕ X[2] ⊕ X[3] ⊕ X[5]⊕ X[6].

C'[2] = X[1] ⊕ X[2] ⊕ X[3] ⊕ X[7].

C'[3] = X[4] ⊕ X[5]⊕ X[6] ⊕ X[7].

C'[4] = X[0] ⊕ X[1] ⊕ X[2] ⊕ X[3] ⊕ X[4] ⊕ X[5]⊕ X[6] ⊕ X[7].

For parity bits prime:

P'L = XL ⊕ XL+8 ⊕ XL+16 ⊕ XL+24 when L is the column number.

1. Pass the old check bits and parity bits to the register without using it.
2. Send these prime bits to the next register.



### Fourth register

Receives the forwarded data, check bits, parity bits, and the newly generated (prime) check and parity bits from the previous stage.

## Fifth stage

Referred to as syndrome generator.

1. Generate syndrome check bits and parity bits by xoring the old bits that we got in encoder with the new bits that we got from prime generator.
2. Xoring the original check bits with check bits prime and store them in new variable called syndrome check bits:

SC[0]= C[0]⊕C'[0].

SC[1]= C[1]⊕C'[1].

SC[2]= C[2]⊕C'[2].

...

...

...

SC[19]= C[19]⊕C'[19].

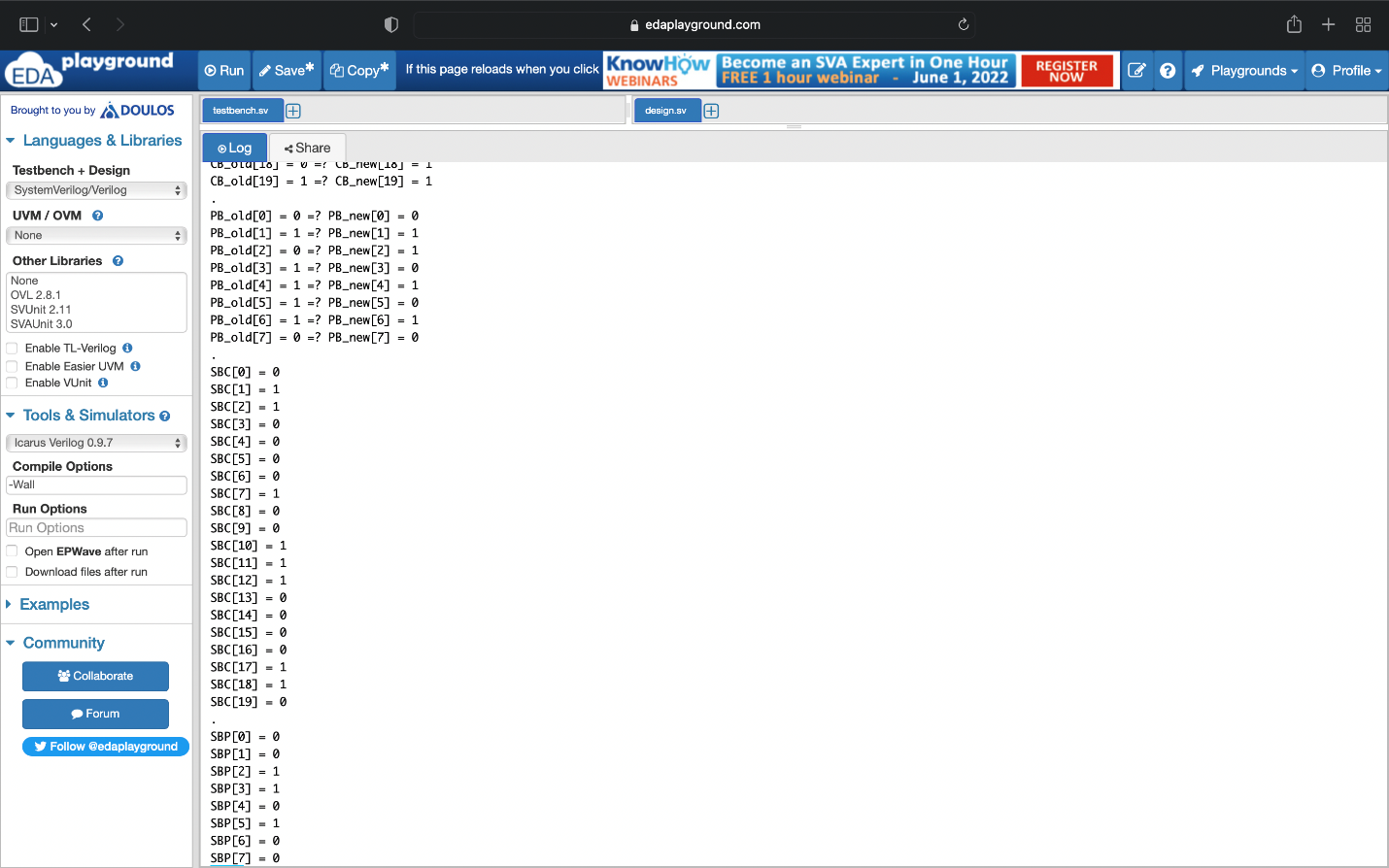
1. Xoring the original parity bits with parity bits prime and store them in new variable called syndrome parity bits:

SP[0]= P[0]⊕P'[0].

…

SP[7]= P[7]⊕P'[7].

1. Pass the data to the register without using it.
2. Send these syndrome bits to the next register.



### Fifth register

Receives the forwarded data, and the syndrome bits from the previous stage.

## Sixth stage

Referred to as identify\_error\_type (Decoder).

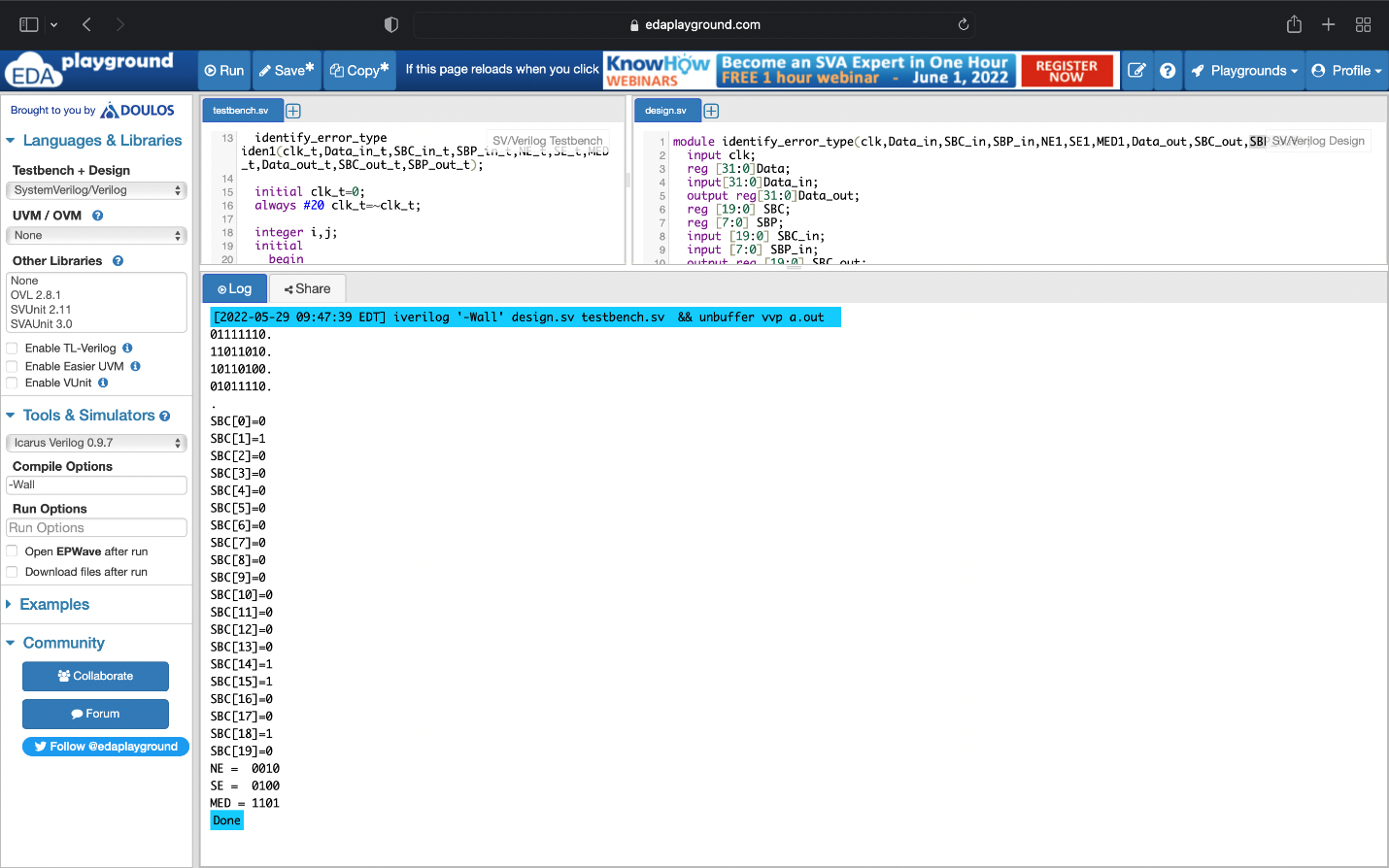
1. This stage is responsible of identify the type of error (if any) has occurred. We have 3 potential results:

* No error (NE).
* Single error detection (SED).
* Multiple error detection (MEP).

1. To generate the SED (single error detection) and NE (no error) signals for each row, we must heck if the syndrome check bit SC(r\*5+4) =1, where r = 0,1,2,3 (row number).

If there is a single error in any row, the corresponding syndrome bit for that row goes high. If none of the syndrome bits are high, then NE signal is generated.

1. Next, generate the MED (Multiple error detection) signal for each row from the syndrome check bits as follows: If (SC(r\*5) OR SC(r\*5+1) OR SC(r\*5+2) OR SC(r\*5+3) OR SC(r\*5+4)) =1 then MEDr =1; where MEDr is the MED signal corresponding to row r.
2. Pass the data and the syndrome bits to the register without using it.
3. Send the generated error signal along with the forwarded data to the next register.



### Sixth register

Receives the forwarded data, syndrome bits, and the error signal from the previous stage.

## Seventh stage

Referred to as the correct\_Errors.

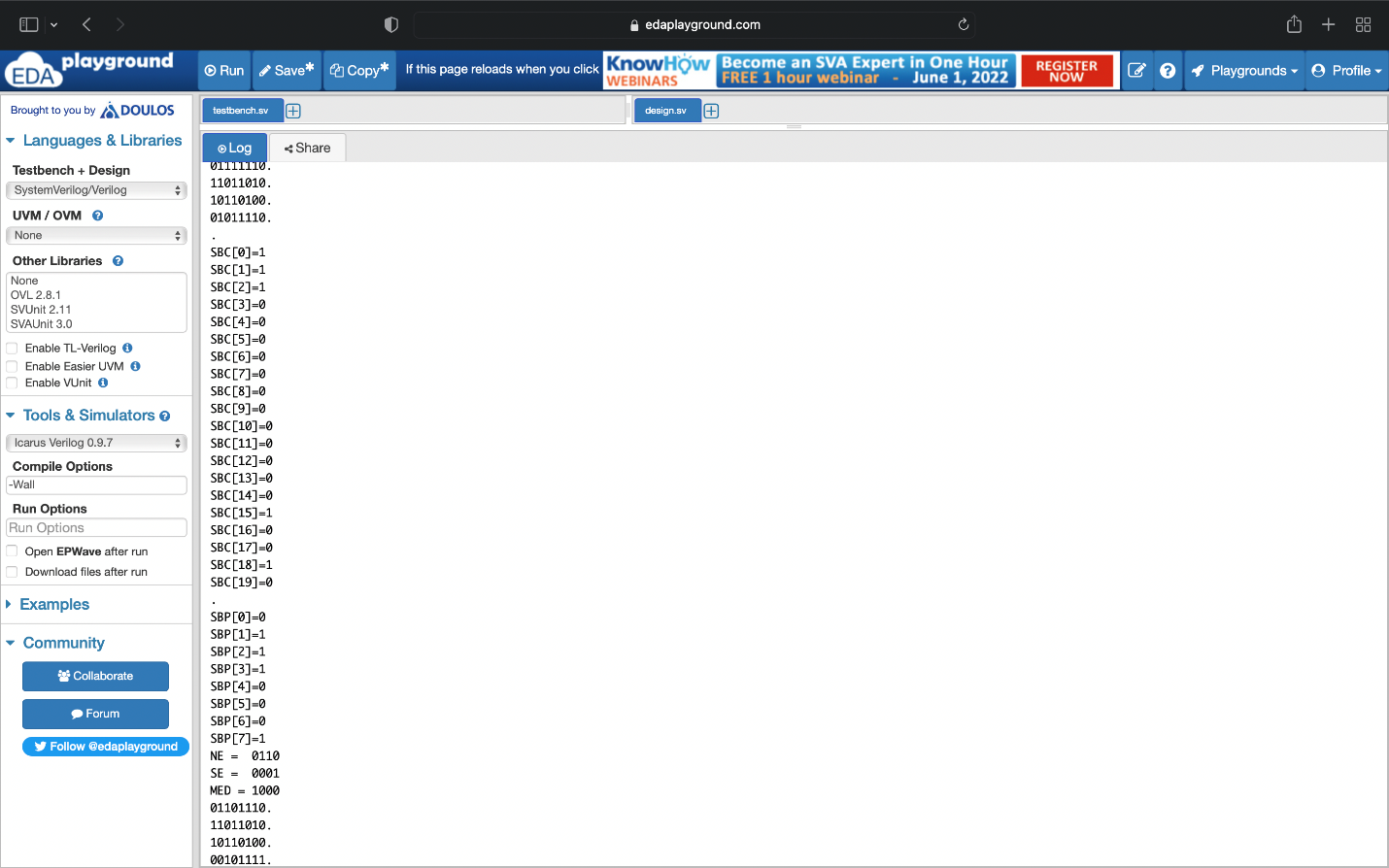
1. Detect the type of error (if any) occurred, depending on the error signal generated and received from the previous stage.
2. Correct the detected single errors using the check syndrome bits as follows:

* If SC0\* SC1\* SC2 =1, then X3 is in error;
* Else if SC0\* SC1\* SC3 =1, then X6 is in error;
* Else if SC0\*SC1 =1, then X0 is in error;
* Else if SC0\* SC2 =1, then X1 is in error;
* Else if SC1\* SC2 =1, then X2 is in error;
* Else if SC1\* SC3 =1, then X5 is in error;
* Else if SC0\* SC3 =1, then X4 is in error;
* Else if SC2\* SC3 =1, then X7 is in error.

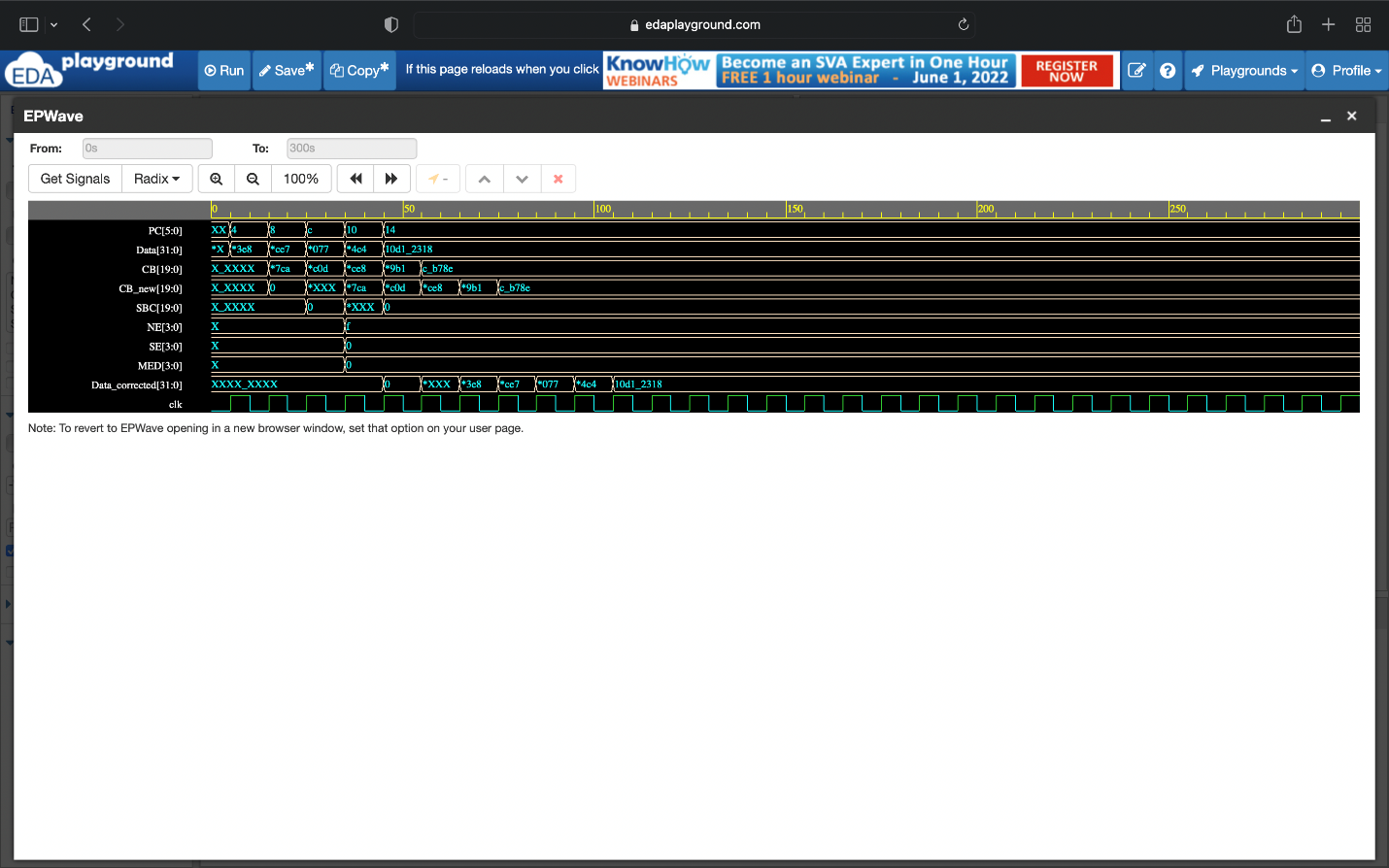
1. Using the parity syndrome bits, correct the multiple errors in a row as follows:

Xicorr = Xi ⊕ ( MEDr \* SPl) where SPl is the syndrome of the parity bit corresponding to the bit l.

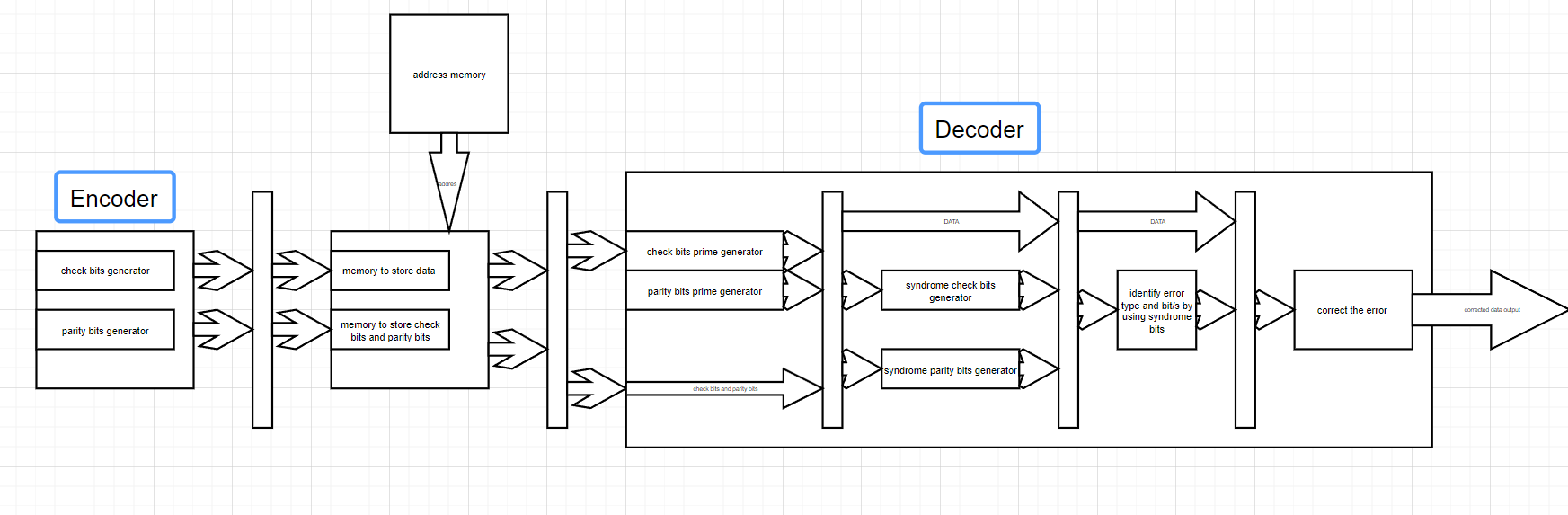
1. Output the corrected 32 bit value.



Final result after testing the pipeline fully:



# Design block diagram



# Design Analysis

This design was simple and pretty straightforward, we didn’t run into design issues as our source for this implementation was written in a clear and concise manner.

Understanding each stage and the method of applying our knowledge of pipelined hardware helped us implement this hardware to the best of our abilities.

# Test runs and discussion

Test runs have been shown at each stage to avoid redundancy in the report.

The output of each stage is shown in the screenshots provided which matches the results expected.

# Performance analysis and comparison with non-pipelined architecture

The pipelined and single-cycled implementation of this design are the same for the first 7 clock cycles.

After the 7th cycle, our pipelined implementation begins to surpass the single-cycled one, as each stage after the 7th, the pipeline is able to output a new corrected 32-bit value.

# Conclusion and future work

By implementing this accelerator using pipelining, we’ve increased the performance and throughput for error detection.

Pipelined implementations are crucial to comprehend and understand, as the current hardware technology is all based on pipelined based structures.

We hope to be able to enhance this error detection and correction technique by applying parallel pipelining techniques.