How To Verify Encoder And Decoder Designs Using Formal Verification

Jin Hou, Mentor-A Siemens Business, Fremont, CA, USA (jin_hou@mentor.com)

I. INTRODUCTION

Encoders and decoders are used a lot in communication systems such as telecommunication, networking, etc. Encoders encrypt data before transmission, and decoders decrypt data at the receiver side. Traditionally encoder and decoder circuits are verified with simulation. Verification engineers have to write lengthy simulation testbenches for generating data vectors, which can be time-consuming. No matter how good the testbenches are, it is impossible to cover all vectors in simulation. Considering the encoder and decoder that can correct random faults, it is even harder for simulation to verify the error-correction function since it cannot cover all errors happening at any bit at any random time. The traditional simulation method is not sufficient for verifying encoders and decoders.

This paper will discuss two new methods based on formal verification for verifying encoders and decoders. One way is using formal property checking, and the other is using formal sequential logic equivalence checking (SLEC). Formal verification is based on a mathematical representation and exhaustive algorithmic techniques. For using formal verification tools, users only need to write simple setup scripts to run formal verification tools and don't need simulation testbenches. Formal tools can automatically consider all possible input values and sequences. By applying the non-deterministic technique, formal tools can consider all random faults to verify the correction function of encoders and decoders.

This paper will use the BCH encoder and decoder IP core in the Opencore public domain as an example to demonstrate how to use two different formal methods to verify encoders and decoders. The BCH encoder and decoder IP core can detect and correct up to 2-bit random errors. When random 1-bit or 2-bit errors happen on the transmission line from ENC to DEC, it should detect the error and set output *error_detected* to 1, and correct the error such that the output data *dout* should be the same as the original data *din*. The structure of the circuit is as follows.

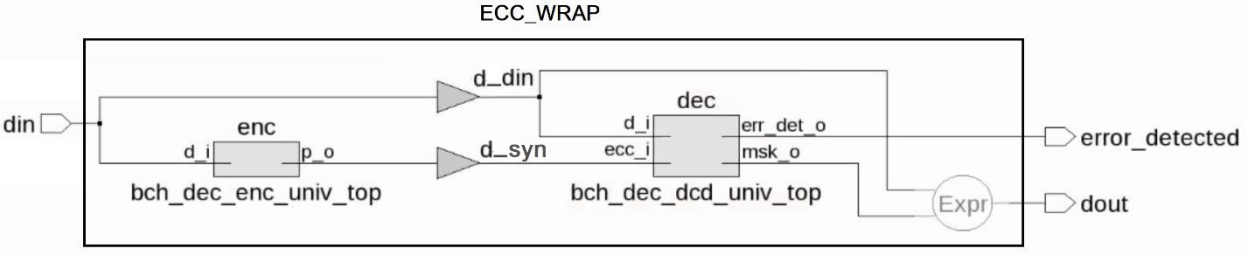

Figure 1. The structure of the BCH encoder and decoder design

II. HOW TO VERIFY BCH ENCODER AND DECODER USING PROPERTY CHECKING

For using property checking, users need to write assertions representing design functions. Property checking verifies the assertions against the design using formal analysis. For verifying the encoder and decoder, verifying them separately from each other is not good since it is hard to express the encryption function and decryption function in assertions. Instead, verifying the overall functions of the circuit is much easier. The properties are listed below.

- When there is no error, *dout* should be the same as *din* and *error_detected* should be 0.
- When 1-bit or 2-bits random errors are happening on the lines to DEC inputs, the errors can be detected, and *error_detected* should be 1.

- When 1-bit or 2-bits random errors are happening on the lines to DEC inputs, the errors can be corrected, and *dout* should be the same as *din*.

One important part of verifying the BCH encoder and decoder circuit is how to represent the random errors happening on the data transmission lines to DEC inputs. Formal tools have a unique feature that they can automatically drive random values to free inputs or undriven wires and also consider all values for them at once. This unique feature is called as Non-Determinism (ND) technique. The ND technique is used to insert random errors.

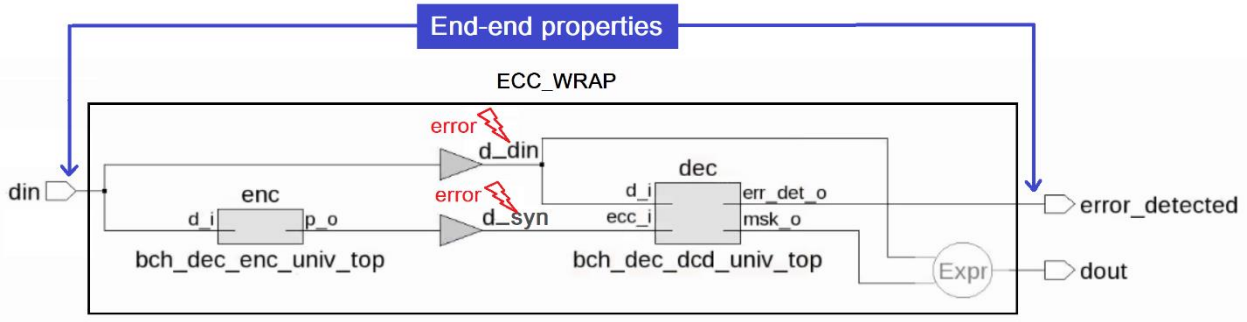

Figure 2. Verifying the BCH ECC design using property checking

An undriven wire *foo* is created for the purpose of controlling the error insertion. Besides, a tool directive *netlist cutpoint* of Questa PropCheck that can force a value to internal signals is used to insert errors under the control of *foo*. When a bit of *foo* is 1, one data transmission line is inserted an error by the tool directive. Each data transmission line is controlled by a corresponding bit of *foo*. The script to insert random errors is in the *insert_errors.do* file as follows.

```
set DATA_WIDTH 16
set ECC_WIDTH 10
set WIDTH [expr $DATA_WIDTH + $ECC_WIDTH]
## Insert random errors to {d_syn, d_din}
for {set i 0} {$i < $WIDTH } {incr i} {
  if {$i < $DATA_WIDTH} {
    netlist cutpoint d_din\[$i\] -cond (foo\[$i\]) -driver ~din\[$i\]
  } else {
    netlist cutpoint d_syn\[[expr {$i-$DATA_WIDTH}]\] -cond (foo\[$i\]) \
      -driver ~e_syn\[[expr {$i-$DATA_WIDTH}]\]
  }
}
```

Figure 3. The *insert_errors.do* file for injecting random errors

The *netlist cutpoint* directive forces the signal slice to the value of the driver only when the condition is true. Figure 4 shows the corresponding change in the circuit when using *netlist cutpoint*. In the above script, the driver is the opposite value of the original signal slice that means the error happens. The condition is *foo[i]* that selects the bit of *d_syn* or *d_din* for injecting a fault. Since *foo* is an undriven wire, formal can consider all of its values at once. Thus all random errors can be inserted into the transmission lines {*d_syn*, *d_din*} and analyzed by formal analysis.

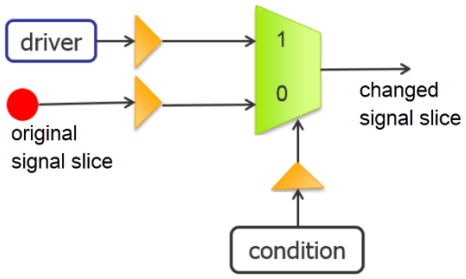

Figure 4. The effect of the *netlist cutpoint* directive

The properties of the encoder and decoder circuit can be written in SVA or Questa PropCheck tool directives. The SVA version is shown below. The number of bits in *foo* that are 1 represents the number of errors inserted. When *$countones(foo)* is 0, no error is inserted, and the *error_detected* signal should be 0. When *$countones(foo)* is 1 or 2, the *error_detected* signal should be 1. When *$countones(foo)* is less or equal to 2, the errors can be corrected such that the *dout* signal should be the same value of *din*.

```
check_no_error: assert property (@($global_clock)
      $countones(foo)==0 |-> ~error_detected);

check_error_detection: assert property (@($global_clock)
      ($countones(foo)==1 || $countones(foo)==2) |-> error_detected);

check_error_correction: assert property (@($global_clock)
      $countones(foo)<=2 |-> dout==din);
```

Figure 5. SVA assertions for verifying the BCH ECC design

The script to run Questa PropCheck is as follows. The *insert_errors.do* file has the tool directives to insert errors.

```
vlog -sv -f qft_files/flist.vl
qverify -c -od log -do " \
do insert_errors.do; \
formal compile -d ecc_wrap;\
formal verify; \
exit"
```

Compile RTL source files.

Read in the tcl file for inserting errors

Build formal model, and run formal analysis

Figure 6. The script to run Questa PropCheck

Questa PropCheck took 20 seconds to finish run for verifying the ECC ciruit for the data width equal to 16 and proved all three properties, i.e., all properties were true for all input data values and all random 1-bit and 2-bit errors

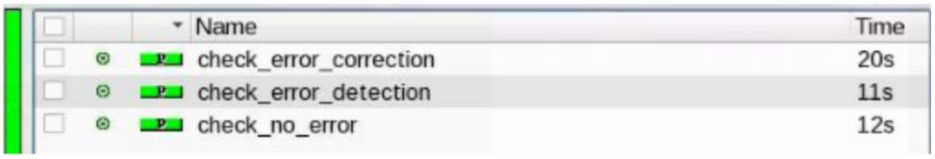

Figure 7. The results from Questa PropCheck

The tool could also generate sanity check waveforms to show that the properties were satisfied. Figure 8 shows the automatic sanity check waveforms for one scenario of the property *check_error_detection*.

Figure 8. The sanity waveforms of *check_error_detection*

The waveforms show *foo==c00000*, that means two bits are 1, and two errors are inserted into the *d_syn* signal. The source tab shows the Verilog code, and *d_syn* is not equal to *e_syn* due to the error injection. From the waveforms and the source tab, we can see that *error_detected* is 1 meaning the ECC circuit can detect the errors, and *dout* is equalent to *din* meaning the ECC design can correct the errors. The tool schematic tab also shows that the ECC circuit is working correctly when two errors are inserted.

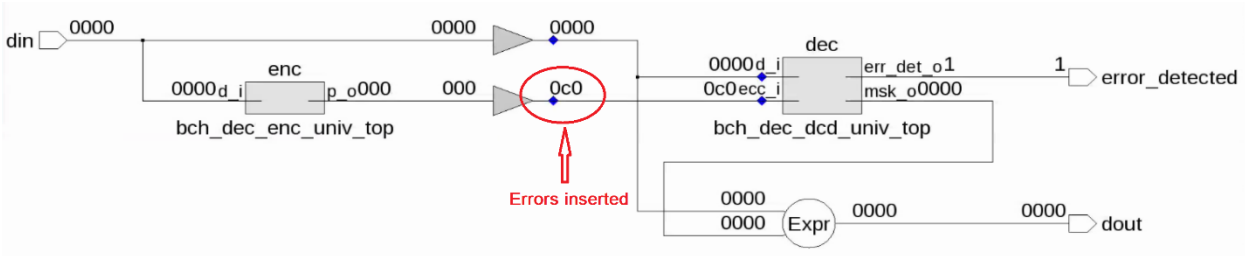

Figure 9. The schematic view associated with the sanity check of *check_error_detection*

The automatic sanity check provides one scenario of the error injection. To see more different error insertions, the user can easily create new cover properties to cover new error insertion scenarios. For example, inside the GUI, the user can click 'New Property' button to open the property editor and create the following cover property such that the tool can insert two errors to the *d_din* signal.

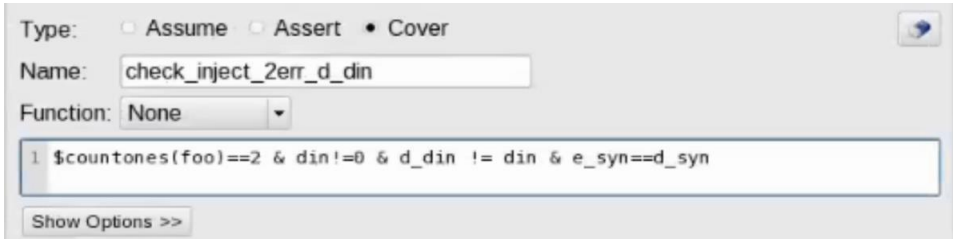

Figure 10. Add cover property to see more error injections

After running the above cover property, the tool generated the following waveforms that inserted two errors to the *d_din* signal.

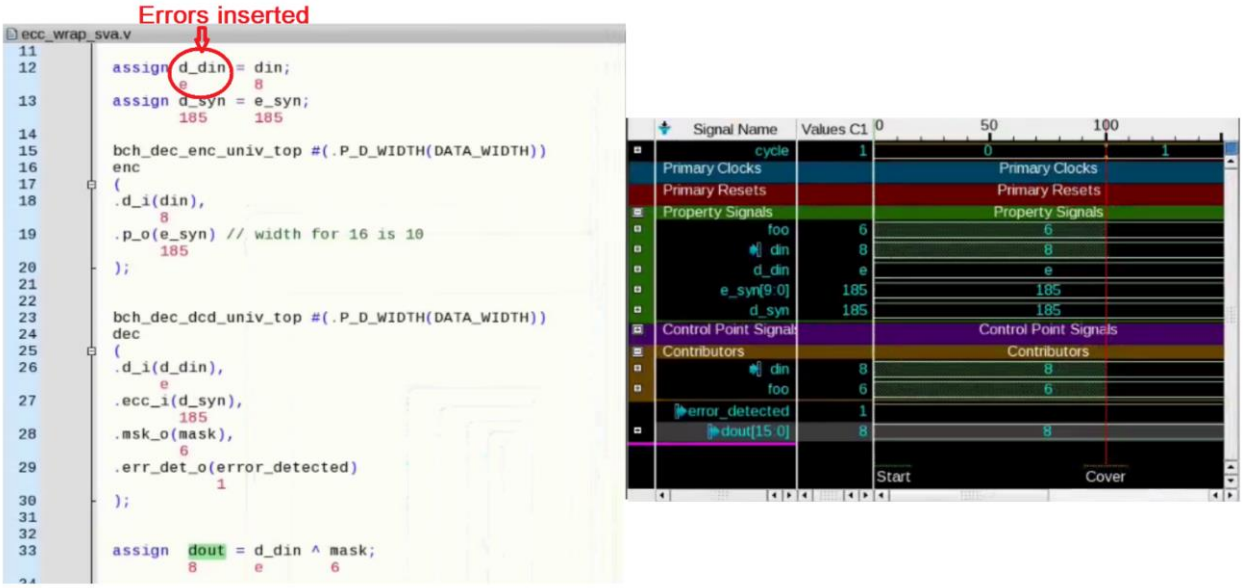

Figure 11. The waveforms and source annotations of new error injection

From the example, we can see that formal property checking can verify encoder and decoder designs. Users need to write executable properties and a simple tool script but don't need to write simulation testbenches. The setup for Questa PropCheck is simple. The formal analysis is exhaustive, and the proof results are for all combinational values of data and random errors of 1-bit/2-bits.

III. How To Verify BCH Encoder And Decoder Using SLEC

Formal method Sequential Logic Equivalence Checking (SLEC) is to verify the functional equivalence of two design versions, i.e., verifying their outputs are always equivalent for any input sequences. When SLEC proves a target (a pair of signals), the targeting pair are equivalent for all possible input scenarios. When SLEC finds one scenario of non-equivalence of targeting pair, it reports the result of *Fired* and shows the scenario in waveforms.

We can use SLEC to compare two versions of the BCH encoder and decoder circuit shown in the following Figure 12. The version *spec* has no error on the transmission lines to DEC inputs, and the version *impl* has random 1-bit or 2-bit errors on the transmission lines to DEC inputs. If the encoder and decoder work correctly, the design should fix 1-bit and 2-bit random errors and detect the error. Thus *dout* should be equivalent to *din*. The *dout* signals of the two design versions should be equivalent. *spec.error_detected* should be zero since *spec* has no error, and *impl.error_detected* should be one since *impl* has 1-bit or 2-bit errors on the transmission lines to DEC and should detect the errors. Thus *spec.error_detected* and *impl.error_detected* are not equivalent.

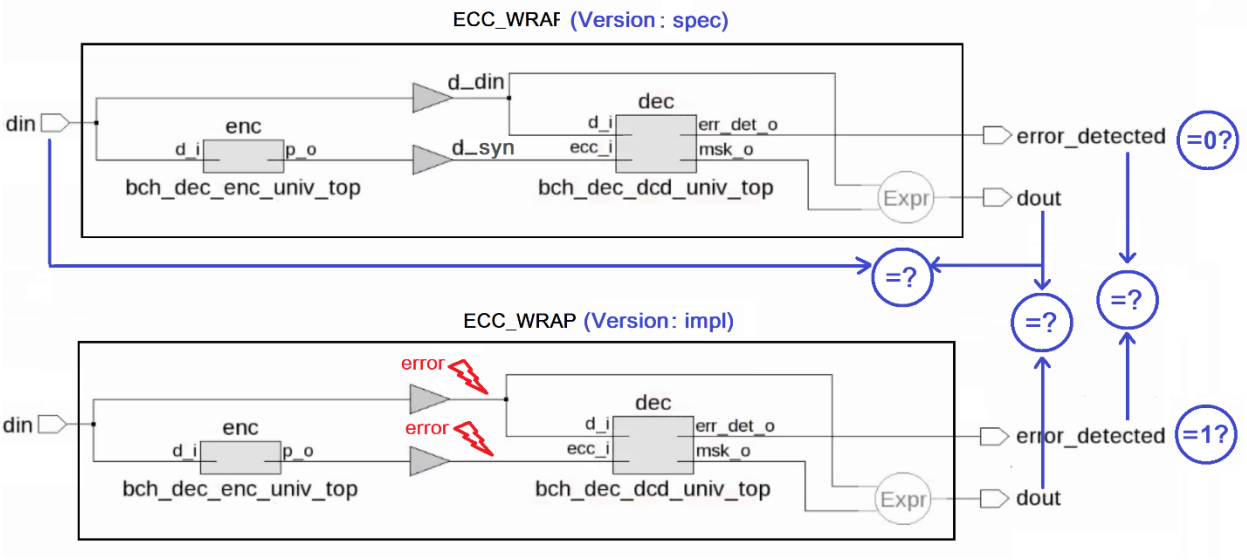

Figure 12. Verifying the BCH ECC design using SLEC

For using SLEC, how to represent all random 1-bit or 2-bit errors is essential. Similar to using property checking, we can use an undriven wire *foo* to control the error insertion. When a bit of *foo* is 1, the tool directive *netlist cutpoint* injects an error into one data transmission line of the version *impl*. Each bit of *foo* controls a corresponding data transmission line of the version *impl*. Since *foo* is undriven wire, formal engines consider all of its values at once. Thus formal analysis considers all random errors that can be inserted into the transmission lines. Since the design only works for 1-bit and 2-bit random errors, to prevent false firings, an assumption to limit the number of 1s in *foo* is needed. The assumption uses the tool directive *netlist property* to constrain *$countones(foo)* to be 1 or 2. The script to insert random errors in the version *impl* is as follows.

```
set DATA_WIDTH 16
set ECC_WIDTH 10
set WIDTH [expr $DATA_WIDTH + $ECC_WIDTH]

## Define undriven wire foo
netlist wire foo -width $WIDTH

## Insert random errors to {d_syn, d_din}
for {set i 0} {$i < $WIDTH} {incr i} {
  if {$i < $DATA_WIDTH} {
    netlist cutpoint impl.d_din\[$i\] -cond {foo\[$i\]} -driver ~spec.d_din\[$i\]
  } else {
    netlist cutpoint impl.d_syn\[[expr {$i-$DATA_WIDTH}]\] -cond {foo\[$i\]} \
             -driver ~spec.d_syn\[[expr {$i-$DATA_WIDTH}]\]
  }
}

## Constrain foo: inject 1 or 2 bit errors
netlist property -name assume_foo -assume {$countones(foo)==1 || $countones(foo)==2}
```

Figure 13. The *inject-errors.do* file for injecting random 1-bit or 2-bit errors to the *impl* version

The script to run Questa SLEC for verifying the BCH ECC design is as follows.

```
vlog -sv -f qft_files/flist.vl
qverify -c -od log_slec -do "\
slec configure -spec -d ecc_wrap;\
slec configure -impl -d ecc_wrap;\
netlist wire const_0 -width 1 -module ecc_wrap -driver 1'b0;\
netlist wire const_1 -width 1 -module ecc_wrap -driver 1'b1;\
do qft_files/inject-errors.do;\
slec map {spec.din impl.dout} -target;\
slec map {spec.const_1 impl.error_detected} -target;\
slec map {spec.error_detected impl.const_0} -target;\
slec compile;\
slec verify;\
exit"
```

Compile RTL source files

Configure spec and impl versions

Define constant signals

Read in the tcl file for inserting errors

Define equivalence targets

Build formal model, create SLEC targets, and run formal analysis

Figure 14. The script to run Questa SLEC

We can see that the script is simple. Questa SLEC automatically ties name-matching inputs of the two versions together to make the inputs equal. It also automatically creates SLEC targets for name-matching outputs. For the BCH ECC design, the tool can automatically generate two targets: one for {*spec.dout impl.dout*} and one for {*spec.error_detectect impl.error_detected*}. Since we also want to verify the equivalency between *dout* and *din* and the equivalency between *error_detected* and constant values, we need to define the SLEC targets using the tool directive *slec map*.

Questa SLEC took 7 seconds to finish run proving four targets and firing one target. The proofs are for all combinations of random *din* and 1-bit/2-bit errors. The proofs are exhaustive. The result is shown below. We can see that *dout* is equal to *din* with or without random 1-bit/2-bit error inserted, which means the design can correct the errors; and *error_detected* is 0 in *spec* when there is no error and is 1 in *impl* when there are random errors, which means the design can detect errors. Thus SLEC has proved the correctness of the encoder and decoder design.

Name	Radiu	Time	Spec Signal	Impl Signal
SLEC_output_2	1	0s	spec.error_detected	impl.error_detected
SLEC_output_1		7s	spec.dout	impl.dout
SLEC_target_1		7s	spec.din	impl.dout
SLEC_target_2		6s	spec.const_1	impl.error_detected
SLEC_target_3		1s	spec.error_detected	impl.const_0
SLEC_input_1			spec.clk	impl.clk
SLEC_input_2			spec.din	impl.din
SLEC_input_3			spec.rstn	impl.rstn

Figure 15. The results from Questa SLEC

The counter example of the target {*spec.error_detected impl.error_detected*} is shown below.

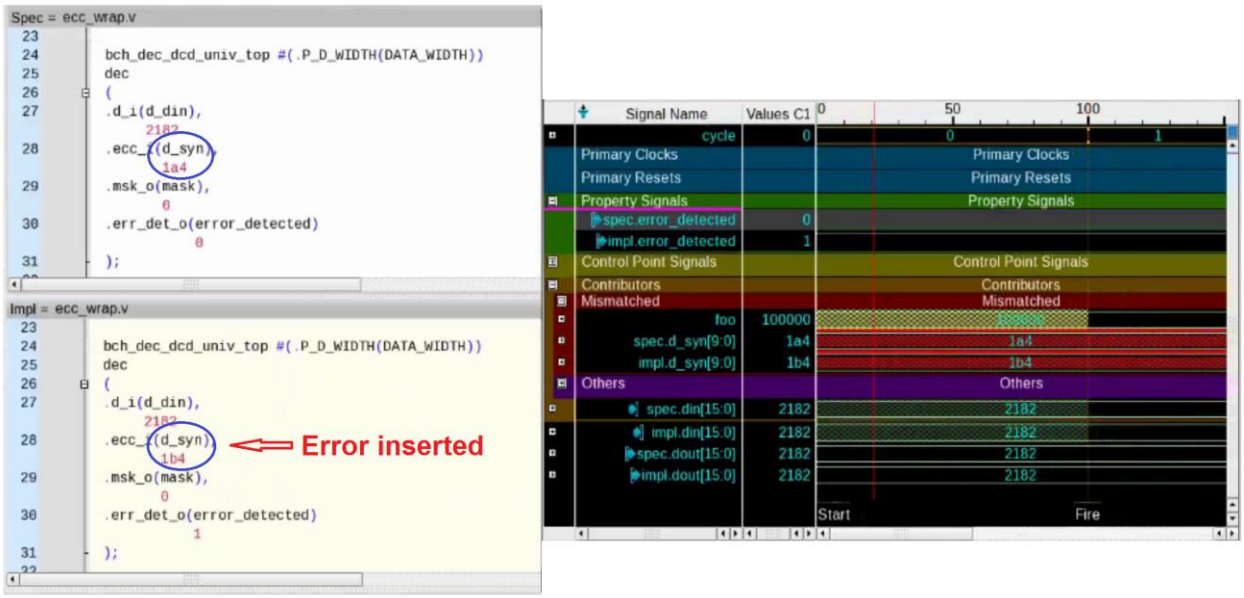

Figure 16. The waveforms of the {*spec.error_detected impl.error_detected*} target

From the waveforms and the source tab, we can see that *spec.din* and *impl.din* are constrained to be equal by the tool automatically. The *impl* version has an error inserted in *d_syn*. The design can detect the fault and set *impl.error_detected* to 1. The design can also correct the error, and the *dout* outputs in both versions are equal to *din*.

From the example, we can see that users don't write SVA assertions when using SLEC, but may define mapping targets when the mapping pair have different signal names. The setup is simple, and we can quickly run the tool to verify the design.

SLEC has lots of different applications [1][3][4][6]. It can verify no bugs injected by low power clock gating, design optimization, or bug fixes/ECO. It can also verify fault tolerance and safety mechanisms, backward compatibility, etc. This paper has discussed one SLEC application. Due to the high degree of automation of SLEC tools, it is worth to try SLEC in your design verification. SLEC is good for both design and verification engineers.

IV. CONCLUSIONS

We have discussed how to use formal property checking and SLEC to verify the BCH encoder and decoder. We have seen the difference when using the two methods. We need assertions when using property checking tools, while we don't need assertions, but need mapping targets of signal pairs for equivalence checking when using SLEC tools. Since Questa SLEC can automatically create targets for name-matching signal pairs, setup for using Questa SLEC is faster without writing assertions. Both Propcheck and SLEC tool finished run in seconds, and proved the design functions.

We have discussed how to use the formal Non-Determinism (ND) technique to consider all random errors. ND is a powerful technique. Using ND effectively with formal tools can simplify the problem and increase the efficiency of formal analysis[2].

For running formal tools, users don't write lengthy simulation testbenches, they only need to write a few simple tool directives. The setup for running formal tools is simple and fast. The big advantage of using formal verification tools to verify encoders and decoders is that formal considers not only all scenarios of input sequences, but also all scenarios of random faults, and can prove design correctness exhaustively.

V. REFERENCES

[1] Jin Hou, Ping Yeung, “Applications of Sequential Logic Equivalence Checking”, DVCon China 2018.

[2] Jin Hou, Mark Eslinger, Ping Yueng, Yuxin You., “Handling Inconclusive Assertions in Formal Verification.” DVCon China 2018

[3] Ping Yueng, Doug Smith, and Abdelouahab Ayari, “Whose fault is it formally? Formal techniques for Optimizing ISO 26262 Fault Analysis,” DVCon San Jose 2018, Feb. 2018

[4] Travis Pouarz, Vaibhav Agrawal, “Efficient and Exhaustive Floating-Point Verification Using Sequential Equivalence Checking”, DCVon San Jose 2017, Feb. 2017.

[5] M Achutha KiranKumar, et al., “Making Formal Property Verification Mainstream: An Intel® Graphics Experience,” DVCon 2017

[6] Adrian Traskov, Thorsten Ehrenberg, et al., “Fault Proof: Using Formal Techniques for Safety Verification and Fault Analysis”, DVCon Europe 2016.

[7] Questa SLEC App https://www.mentor.com/products/fv/questa-slec

[8] Questa SLEC Course on Verification Academy. https://verificationacademy.com/courses/sequential-logic-equivalence-checking