How To Verify Encoder And Decoder Designs Using Formal Verification

Jin Hou
Mentor-A Siemens Business
Agenda

• Introduction
• How to verify BCH encoder and decoder using property checking
• How to verify BCH encoder and decoder using SLEC (Sequential Logic Equivalence Checking)
• Conclusions
Introduction

• The challenges for fully verifying encoder and decoder designs:
  – How to cover all possible data coming into the encoder
  – How to cover all possible random faults if the design should correct them?

• A case study: BCH encoder and decoder design from Opencore website.
  – The function of BCH: Fix up to 2-bit random faults injected to the transmission lines from the encoder to the decoder.
Verifying BCH Design Using Property Checking

• Property checking flow:

• The functions of BCH
  1. When there is no error, \( dout \) should be the same as \( din \) and \( error\_detected \) should be 0.
  2. When 1-bit or 2-bit random errors are happening on the lines to DEC inputs, the errors can be detected, and \( error\_detected \) should be 1.
  3. When 1-bit or 2-bit random errors are happening on the lines to DEC inputs, the errors can be corrected, and \( dout \) should be the same as \( din \).
Inserting Random Faults

• Define an undriven wire `foo`. When a bit of `foo` is 1, Questa PropCheck directive `netlist cutpoint` inserts an error to one data transmission line.

• Formal verification automatically considers all possible values for the undriven wire `foo`.

```
set DATA_WIDTH 16
set ECC_WIDTH 10
set WIDTH [expr $DATA_WIDTH + $ECC_WIDTH]
for {set i 0} {$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\[[expr {$i-$DATA_WIDTH}]\] }
    }
```
Writing Properties In SVA

• Translate the properties to executable SVA assertions.
  – *foo* controls the number of faults injected to the transmission lines between the encoder and the decoder

```sva
check_no_error: assert property (@($global_clock)
    foo==0 |-> ~error_detected);
check_error_detection: assert property (@($global_clock)
    ($countones(foo)==1 || $countones(foo)==2) |-> error_detected);
check_error_correction: assert property (@($global_clock)
    $countones(foo)<=2 |-> dout==din);
```
Verifying BCH Design With Questa PropCheck

• The script to run Questa PropCheck is as follows. The `insert_errors.do` file inserts random 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"
```

• The verification result:

![Verification Result](image-url)
Sanity Waveforms

- The sanity waveforms of the property `check_error_detection`
The Schematic View

- The Schematic View shows the inserted faults.
The Flow Of SLEC

- The flow of SLEC (Sequential Logic Equivalent Checking)
  - The SLEC tool automatically maps the inputs of DUT0 and DUT1 and ties them together
  - The SLEC tool automatically maps the outputs of DUT0 and DUT1 and verifies their equivalency.
Verifying BCH Design Using SLEC

- The design version `spec` is the BCH design without faults
- The design version `impl` is the BCH design with random faults
Verifying BCH Design With Questa SLEC

• The script to run Questa SLEC to verify BCH design:

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

- 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
Inserting Random Faults

- Inject-errors.do file:

```bash
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}
```
The Verification Results With Questa SLEC

- The verification results:

<table>
<thead>
<tr>
<th>Name</th>
<th>Radix: Time</th>
<th>Spec Signal</th>
<th>Impl Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEC_output_2</td>
<td>1</td>
<td>spec.error_detected</td>
<td>impl.error_detected</td>
</tr>
<tr>
<td>SLEC_output_1</td>
<td>7s</td>
<td>spec.dout</td>
<td>impl.dout</td>
</tr>
<tr>
<td>SLEC_target_1</td>
<td>7s</td>
<td>spec.din</td>
<td>impl.dout</td>
</tr>
<tr>
<td>SLEC_target_2</td>
<td>6s</td>
<td>spec.const_1</td>
<td>impl.error_detected</td>
</tr>
<tr>
<td>SLEC_target_3</td>
<td>1s</td>
<td>spec.error_detected</td>
<td>impl.const_0</td>
</tr>
<tr>
<td>SLEC_input_1</td>
<td></td>
<td>spec.clk</td>
<td>impl.clk</td>
</tr>
<tr>
<td>SLEC_input_2</td>
<td></td>
<td>spec.din</td>
<td>impl.din</td>
</tr>
<tr>
<td>SLEC_input_3</td>
<td></td>
<td>spec.rstn</td>
<td>impl.rstn</td>
</tr>
</tbody>
</table>
Counterexample Of Non-equivalence

• The counterexample of the target \{\text{spec.error\_detected impl.error\_detected}\}
Conclusions

• Both property checking and SLEC can exhaustively verify BCH encoder and decoder design
  – Property checking verifies assertions against the design.
  – SLEC verifies the equivalency between the versions with/without injected faults.
  – Formal algorithms consider all possible inputs and random faults.
    • Simulation is lacking of this capability, and cannot exhaustively prove design functions.

• Setup is easy and fast for both formal methods
  – No lengthy simulation testbenches.

• Formal verification methods property checking and SLEC can be applied to other encoder and decoder designs.
Questions