Synthesis of Decoder Tables using Formal Verification Tools

Keerthikumara Devarajegowda, Johannes Schreiner and Wolfgang Ecker
Introduction

Designs

Controller based
Non-controller based
Introduction

What we mean by **Controller based designs**?
Introduction
Introduction

Controller logic

micro-architecture
Introduction

- Typical instruction decoder of a processor
- Decoding is generally implemented using a lookup-table
State-of-the-Art : Manual coding

- Design engineer computes control signals

- Requires deep knowledge of the micro-architecture

- Not suitable for late specification changes/extensions

- Often repetitive
(NOT) State-of-the-Art: Simulation tech

- Simulation technique can be used to determine control signals
- Repetitive
- Time-tedious
- Not suitable for medium/large set of control signals
State-of-the-Art: Simulation tech

- Simulation technique can be used to determine control signals
  - Repetitive
  - Time-tedious
  - Not suitable for medium/large set of control signals

What about Formal verification tools?
Our idea for synthesis of control signals

Specification

Properties
(SVA)

Micro – Architecture with empty lookup table
Our idea for synthesis of control signals

Specification ➔ Properties (SVA) ➔ Formal Verifier

Micro – Architecture with empty lookup table
Our idea for synthesis of control signals

Specification → Properties (SVA) → Formal Verifier → Witness Trace

Specification with Micro – Architecture with empty lookup table
Our idea for synthesis of control signals

Specification -> Properties (SVA) -> Formal Verifier -> Witness Trace

Micro-Architecture with empty lookup table -> Look-up Generator

Control-signal
First Try…

- We simply used `COVER` directive to extract the decoder settings/control signals
- Did not work!
- The formal tool returned wrong decoder settings
First Try…

- We simply used **COVER** directive to extract the decoder settings/control signals
- Did not work!
- The formal tool returned wrong decoder settings

Simple example:

\[
\begin{align*}
0 + 0 &= 0 & \text{ADD} \\
0 - 0 &= 0 & \text{SUB}
\end{align*}
\]
Second Try…

• Next, we used `ASSERT` directive with negation

• i.e., $A \rightarrow B$ is transformed to $A \rightarrow \text{not } B$

• Since decoder is empty, the tool shall give a `CEX` for $A \rightarrow B$
Next, we used **ASSERT** directive with negation

**i.e.,** $A \rightarrow B$ is transformed to $A \rightarrow \neg B$

Since decoder is empty, the tool shall give a **CEX** for $A \rightarrow B$

But, **De Morgan’s law**

**Assume** $B = b_1$ and $b_2$ and $b_3$

$\neg B = \neg b_1 \text{ or } \neg b_2 \text{ or } \neg b_3$

**OR** – operation allows alternative settings
Solution!

• Finally, we extracted the correct control signals by excluding other operations

• $A \rightarrow (B \text{ and } \neg C)$
Solution!

• Finally, we extracted the correct control signals by excluding other operations

• $A \rightarrow (B \text{ and } \neg C)$

• Simple example: $(out = in1 + in2)$ and $(out \neq in1 - in2)$

• The $COVER$ directive shall give correct control values

• Since we generate the properties, generating exclusion is not a big effort
Case Study: 5-stage RISCV Core

- 5-stage RISCV core with an empty lookup table in the instruction decoder
- Semi-automated approach
- Python based in-house automation framework
Case Study: 5-stage RISCV Core

- SVA: Similar approach (as RTL) for Property generation
- Python based automation framework
- A property for every instruction
- Property captures the required signal behavior of the micro-architecture
- Later re-used for verification

```verilog
1   property _ADD;
2       @(posedge clk) disable iff(reset)
3          assume_add_instr
4          |->
5            ##0
6       rs1_addr == instr[19:15] &&
7       rs2_addr == instr[24:20] &&
9            ##1
10      alu_in1 == $past(rs1_data) &&
11      alu_in2 == $past(rs2_data) &&
12      alu_result == alu_in_1 + alu_in2 &&
13            exclude_other_ops
14            ##1
15       !data_mem_wr_en
16            ##1
17      reg_wr_data == $past(alu_result,2) &&
18      reg_wr_en &&
19      reg_wr_addr == $past(instr[11:7],3);
20    endproperty
21   COV_add_instr: cover property(_ADD);
```
Case Study: 5-stage RISCV Core

```verilog
define property _ADD;
@(posedge clk) disable iff(reset)
  assume_add_instr
  ->
  ##0
    rs1_addr == instr[19:15] &&
    rs2_addr == instr[24:20] &&
  ##1
    alu_in1 == $past(rs1_data) &&
    alu_in2 == $past(rs2_data) &&
    alu_result == alu_in_1 + alu_in2 &&
    exclude_other_ops
  ##1
    !data_mem_wr_en
  ##1
    reg_wr_data == $past(alu_result,2) &&
    reg_wr_en
    &&
    reg_wr_addr == $past(instr[11:7],3);
endproperty

COV_add_instr: cover property(_ADD);
```

alu_result != (alu_in1 >> alu_in2[4:0]) &&
alu_result != (alu_in1 & alu_in2)) &&
alu_result != (alu_in1 << (alu_in2[4:0])))&&
alu_result != (alu_in1 - alu_in2)) &&
alu_result != (alu_in1 < alu_in2)) &&
alu_result != (alu_in1 | alu_in2))
Scripting to automate extraction

• We used formal tool **Onespin 360** for our work
• But our approach **shall work with all formal tools**

• The tool allows to select specific signal values at specific time-points
• Script to parse the log file and extract control signal values
• Supplement RTL generator with required control signals
Conclusion

 ✓ Formal methods are powerful techniques and shall be used to support design development as well
 ✓ Significant reduction of boring and error prone manual efforts
 ✓ Developed SVAs are re-used for verification
 ✓ Quick turn-around time for late changes and/or extensions
 ✓ The approach is applicable for wide-variety of problems as soft and hard configuration of modules
Thank you!