The Application of Formal Technology on Fixed-Point Arithmetic SystemC Designs

Sven Beyer, Dominik Straßer, Dave Kelf
OneSpin Solutions GmbH
first.last@onespin-solutions.com
Agenda

• Intro
  – SystemC Flow
  – Floating/Fixed Point Arithmetic

• Formal Verification on SystemC
  – Automatic Fixed Point Verification
  – SVA assertions
  – Design exploration
SystemC HLS Flow

**C++/SystemC**
- C/C++ algorithm executable specification
- C++/SystemC Loosely Timed MicroArchitecture
- SystemC Cycle Accurate Model

**Verilog/VHDL**
- High Level Synthesis (HLS)
- Verilog / VHDL RTL
  - RTL Synthesis P&R
  - Gate P&R

**MATLAB SIMULINK**
C/C++ Algorithms

• Use IEEE 754 floating point numbers
• Cover wide range of numbers with “good” precision
• Ideal for software due to CPU hardware support
• Floating point hardware very complex - see FDIV bug 1995
• Floating point algorithms not synthesizable
Fixed Point Arithmetic

• sign + n-bit binary value (like signed Verilog types)
• additional m bits binary fraction
• Bit value \( a[i] \times 2^i \)
  - First fractional bit valued 0.5, then 0.25, ...

• \( n+m \) bits precision without scaling exponent
• Hardware basically just integer hardware
Float vs. Fixed

<table>
<thead>
<tr>
<th>64-bit float</th>
<th>64-bit fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>53 bits precision (mantissa)</td>
<td>63 bits precision</td>
</tr>
<tr>
<td>11 bits exponent for scaling</td>
<td>-</td>
</tr>
<tr>
<td>Complex hardware</td>
<td>Simple hardware</td>
</tr>
</tbody>
</table>

- Fixed may actually be more precise due to 10 bits added precision
- Fixed “good enough” for numbers in specific range
- Synthesizable, fully templatized fixed point classes with overloaded operators in SystemC
- Need “right” number of bits before/after.
Example: FIR Filter with Fixed Point
Formal Verification

Question about DUT ("assertion")

Apply on source code to find simulation traces

Formal Verification

Apply on source code

C++/SystemC

C/C++ algorithm executable specification

C++/SystemC Loosely Timed MicroArchitecture

OR

SystemC Cycle Accurate Model

High Level Synthesis (HLS)

or prove that none exist
FIR Filter in Debugger

Values in original type

```c
while (true) {
    result.write(
        1.1143441535532474517822265625
        -0.07556556070608 * sample.read() +
        .466072373092174530029296875 -> .3443153770640173967
        .09129209297815 * pipe0 +
        -1.043895117938518524169921875 -> .466072373092174530029296875
        0.47697917208036 * pipel +
        1.2999389879405498504638671875 -> -1.043895117938518524169921875
        0.47697917208036 * pipe2 +
        .2950866706669330596923828125 -> .12999389879405498504638671875
    );
}
```
Automated formal analysis

• Generated “assertions” to check for
  – Arithmetic overflow
    • Does individual operation produce overflow?
  – Redundant bits
    • Is MSB of unsigned fixed float always 0?
    • Are 2 MSBs of signed fixed float always equal?
• Prove “right” number of fixed float bits formally
Redundant Bits

Session Setup File Edit CC/MV EC Tools Window Help

Design Explorer Auto Checks Dead-Code Checks Assertion

Design View

Path: [top]

```cpp
while (true) {
    if (start.read()==1) {
        result.write(op_a.read()*0.15+op_b.read()*0.34);
        done.write(1);
    } else {
    }
}
```

compute.cpp (read-only view) line 4, column 1

Shell

```
-I- Size of integer_check 'integer_check_1.bit_4': Vars(< 20) Nodes(< 1000)
-I- Size of integer_check 'integer_check_1.bit_5': Vars(< 20) Nodes(< 1000)
-I- Size of integer_check 'integer_check_1.bit_6': Vars(< 20) Nodes(< 1000)
-V- AutoCheck - 'integer_check_1': 1 redundant bits in result
```

1 redundant bit identified
Overflow Detection

```c
while (true) {
    if (start.read()==1) {
        true->false
        sc_fixed<8,2> lop_a = op_a.read()*0.75;
        -1.5->0 -1.98438->0
        sc_fixed<8,2> lop_b = op_b.read()*0.34;
        -0.515625->0 -1.51562->0
        result.write(lop_a + lop_b);
        0->1.98438 -1.5->0 -0.515625->0
        done.write(1);  
        false->true
    } else {
        done.write(0);
    }
}
```
SVA assertions on SystemC

• SVA allows to “bind” monitors to Verilog and VHDL
• Additional support for SystemC allows full-fledged SVA support on top of SystemC
  – Temporal assertion with sequences of interesting values
  – Liveness assertions
  – Requires SVA extension to support fixed point data types
• Derive assertions from specification to automatically
  – proves absence of failures or
  – Finds corner case failures
Interactive Formal Analysis

• Express interesting sequence of output values in SVA

Formal tool finds input sequence producing desired output sequence
Summary

• Formal verification of SystemC with fixed float types
  – Automatic checks for redundant bits and overflows
• Full SVA support on SystemC
  – Extension for fixed float types in SVA
  – Design exploration with interesting sequences of outputs
  – Assertion development from spec for formal verification
• All verification and debugging on original SystemC using high level data types like fixed float
Questions?