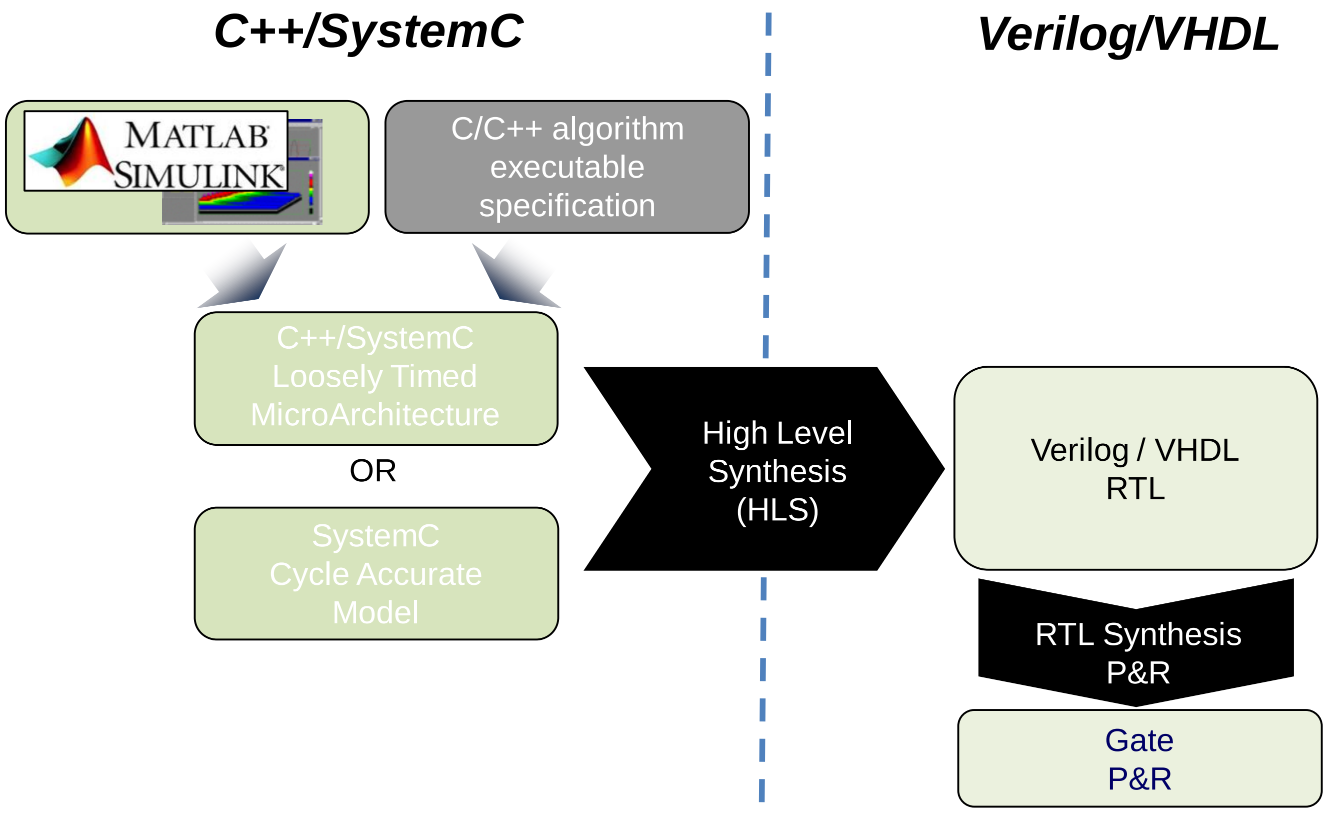
The Application of Formal Technology on Fixed-Point Arithmetic SystemC Designs

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

Agenda

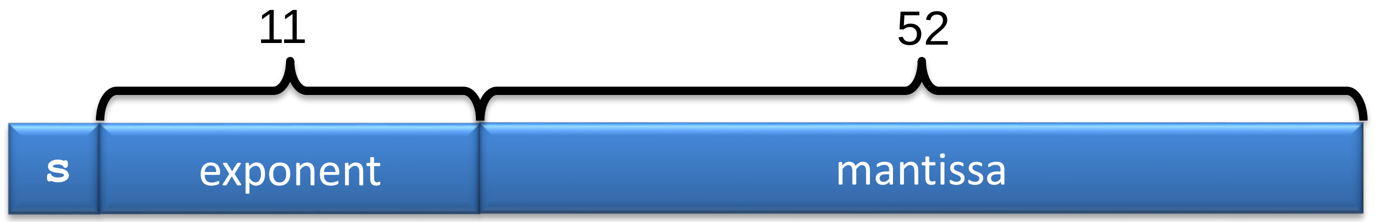
- Intro
  - SystemC Flow
  - Floating/Fixed Point Arithmetic
- Formal Verification on SystemC
  - Automatic Fixed Point Verification
  - SVA assertions
  - Design exploration

SystemC HLS Flow


C++/SystemC
Verilog/VHDL
MATLAB SIMULINK
C/C++ algorithm executable specification
C++/SystemC Loosely Timed MicroArchitecture
OR
SystemC Cycle Accurate Model
High Level Synthesis (HLS)
Verilog / VHDL RTL
RTL Synthesis P&R
Gate P&R

C/C++ Algorithms

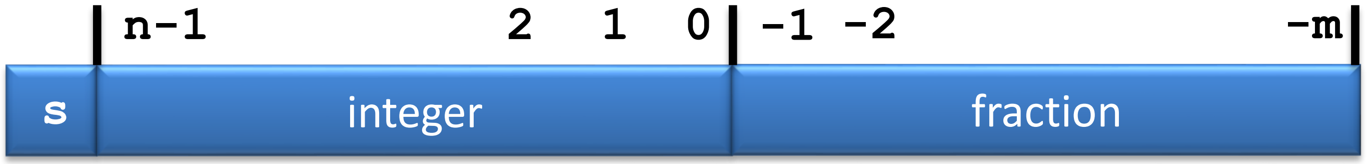
- Use IEEE 754 floating point numbers



- 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] * 2^i`**
  - First fractional bit valued 0.5, then 0.25, …



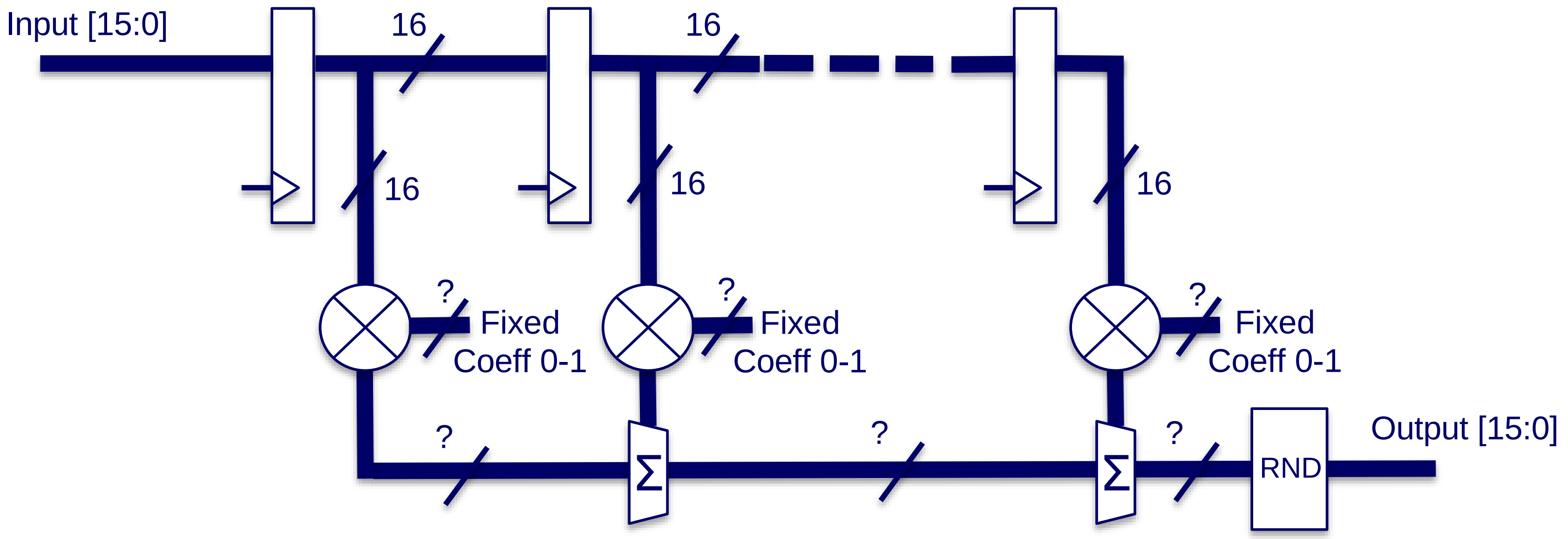
- **`n+m`** bits precision without scaling exponent
- Hardware basically just integer hardware

Float vs. Fixed

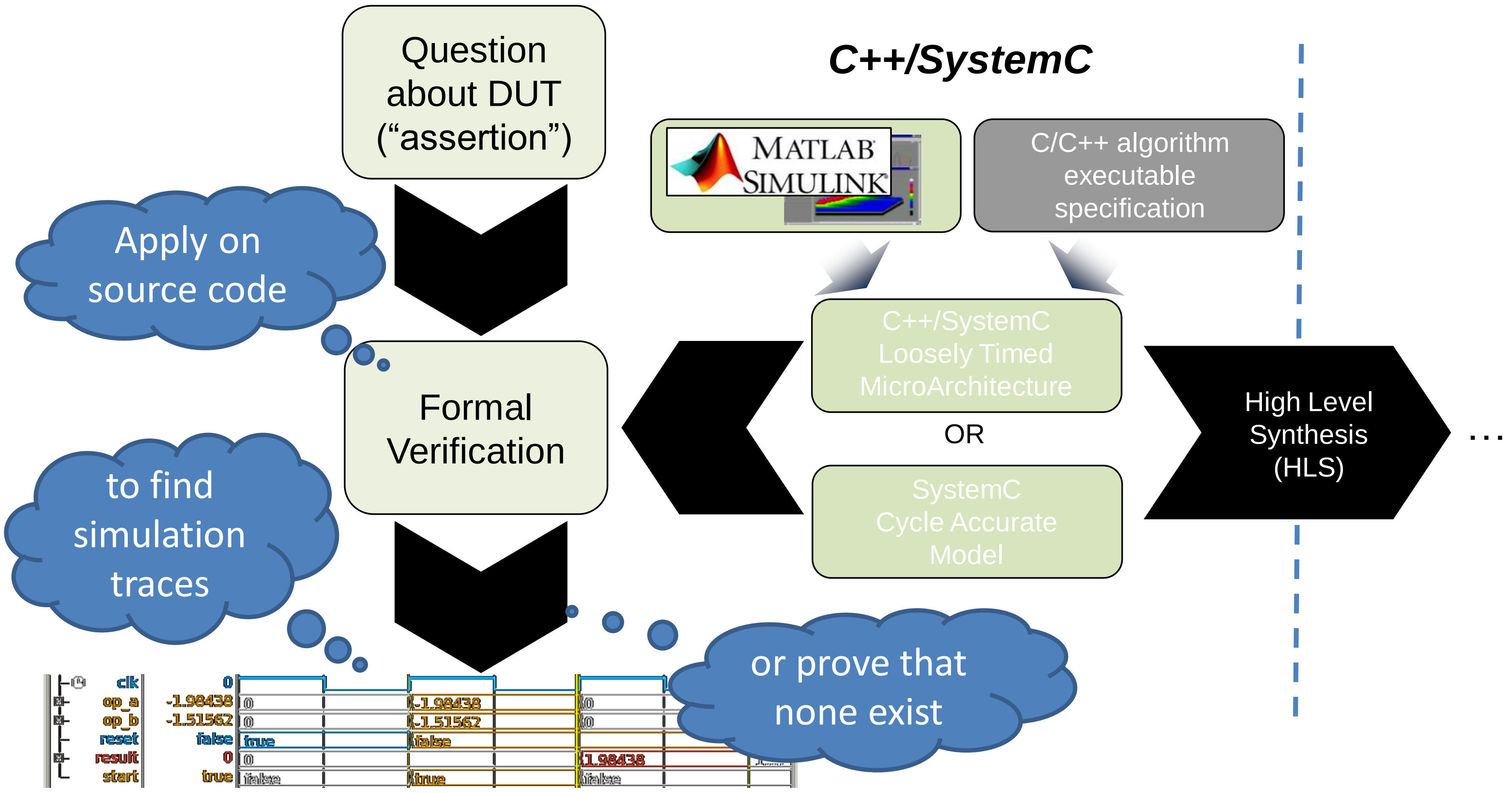
64-bit float	64-bit fixed
53 bits precision (mantissa)	63 bits precision
11 bits exponent for scaling	-
Complex hardware	Simple hardware

- 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

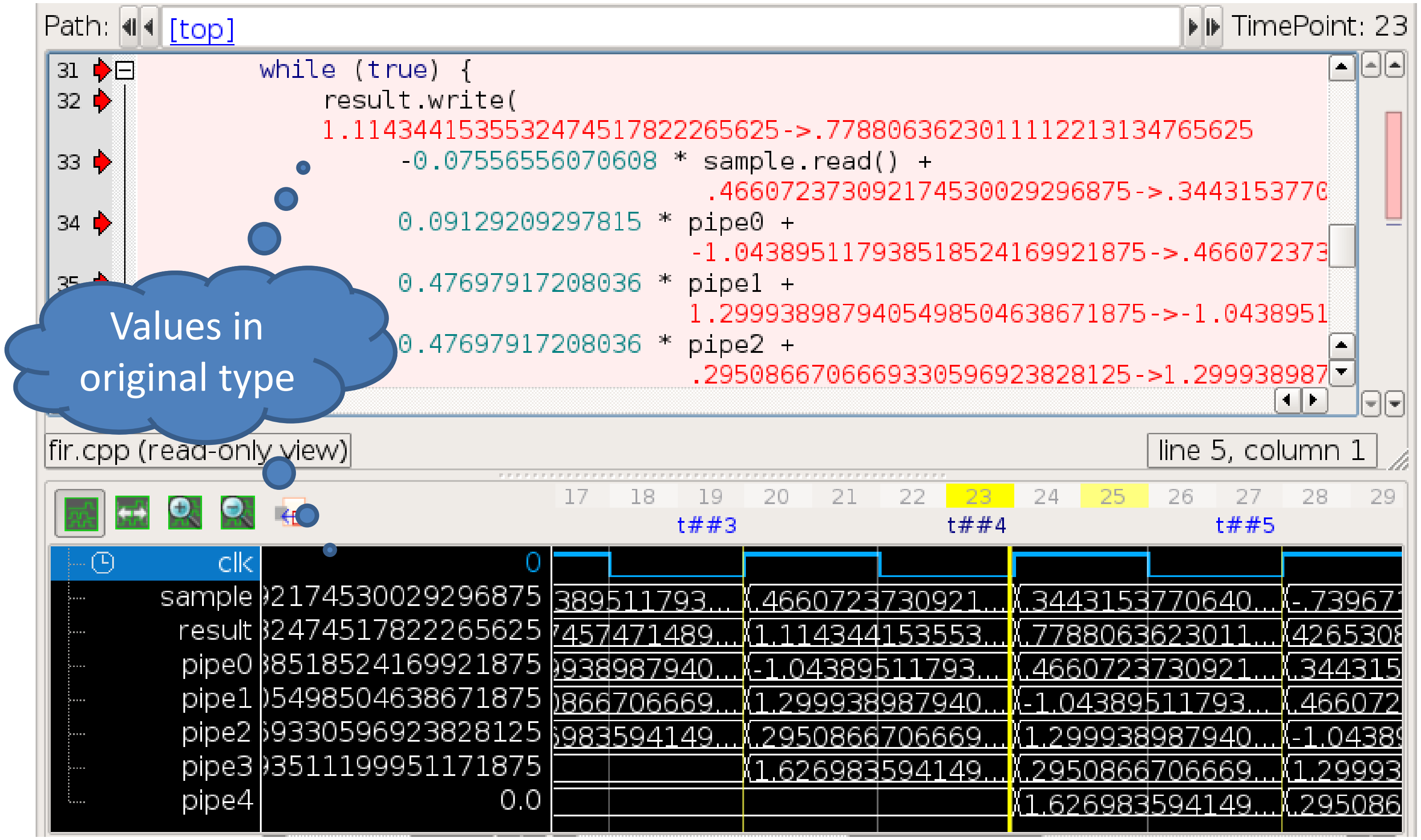


Formal Verification



Question about DUT ("assertion")
Apply on source code
Formal Verification
to find simulation traces
or prove that none exist
C++/SystemC
MATLAB SIMULINK
C/C++ algorithm executable specification
C++/SystemC Loosely Timed MicroArchitecture
OR
SystemC Cycle Accurate Model
High Level Synthesis (HLS)
…
clk 0
op_a -1.98438
op_b -1.51562
reset false
result 0
start true

FIR Filter in Debugger



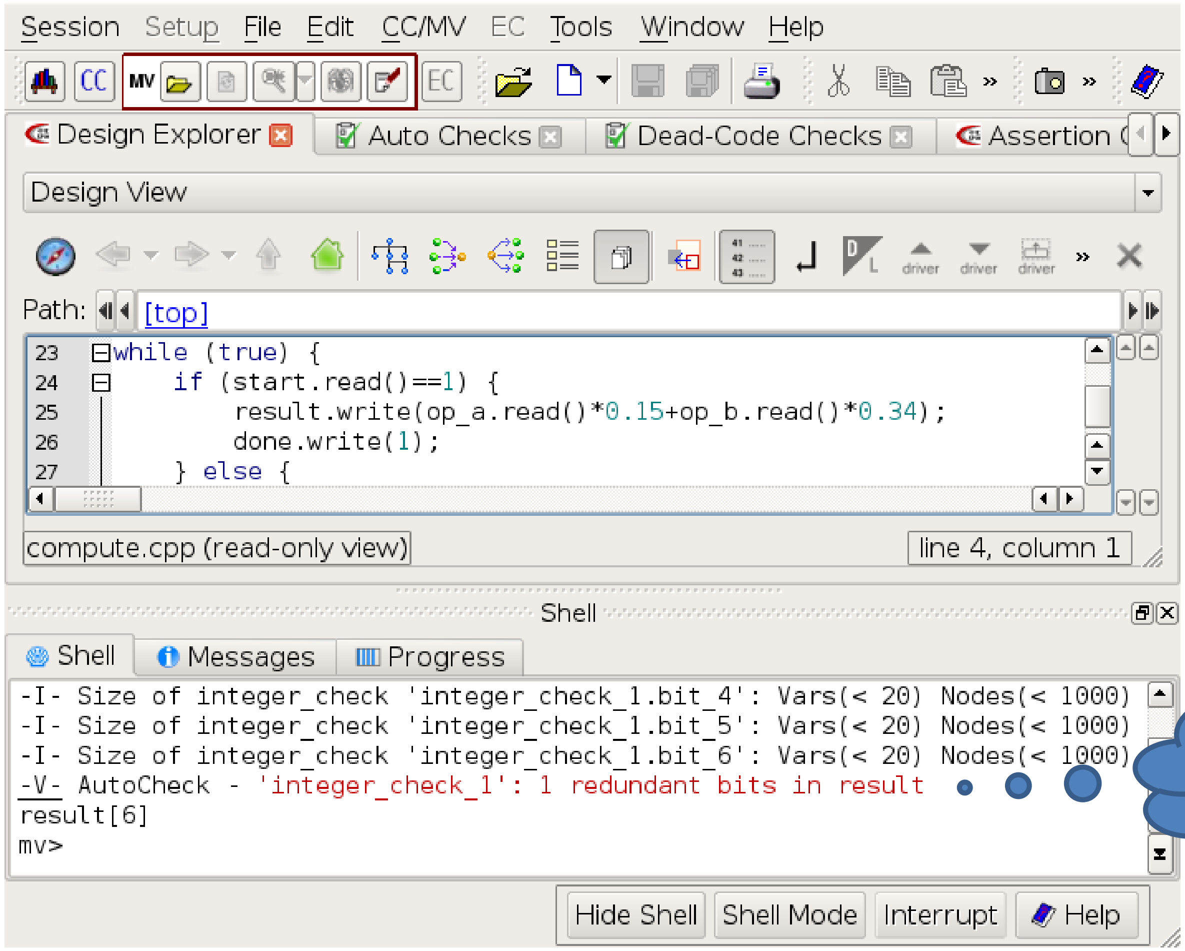
```
        while (true) {
            result.write(
            1.1143441535532474517822265625->.778806362301112213134765625
                -0.07556556070608 * sample.read() +
                                    .466072373092174530029296875->.3443153770
                0.09129209297815 * pipe0 +
                                   -1.043895117938518524169921875->.466072373
                0.47697917208036 * pipe1 +
                                   1.2999389879405498504638671875->-1.0438951
                0.47697917208036 * pipe2 +
                                   .2950866706669330596923828125->1.299938987
```

```
                 17   18   19   20   21   22   23   24   25   26   27   28   29
                          t##3                t##4                t##5
clk                         0
sample   )2174530029296875  389511793...  .4660723730921...  .3443153770640...  -.739671
result   )2474517822265625  457471489...  1.114344153553...  .7788063623011...  4265308
pipe0    )8518524169921875  9938987940...  -1.04389511793...  .4660723730921...  .344315
pipe1    )5498504638671875  )866706669...  1.299938987940...  -1.04389511793...  .466072
pipe2    )9330596923828125  )983594149...  .2950866706669...  1.299938987940...  -1.04389
pipe3    )3511199951171875                 1.626983594149...  .2950866706669...  1.29993
pipe4                   0.0                                    1.626983594149...  .295086
```

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

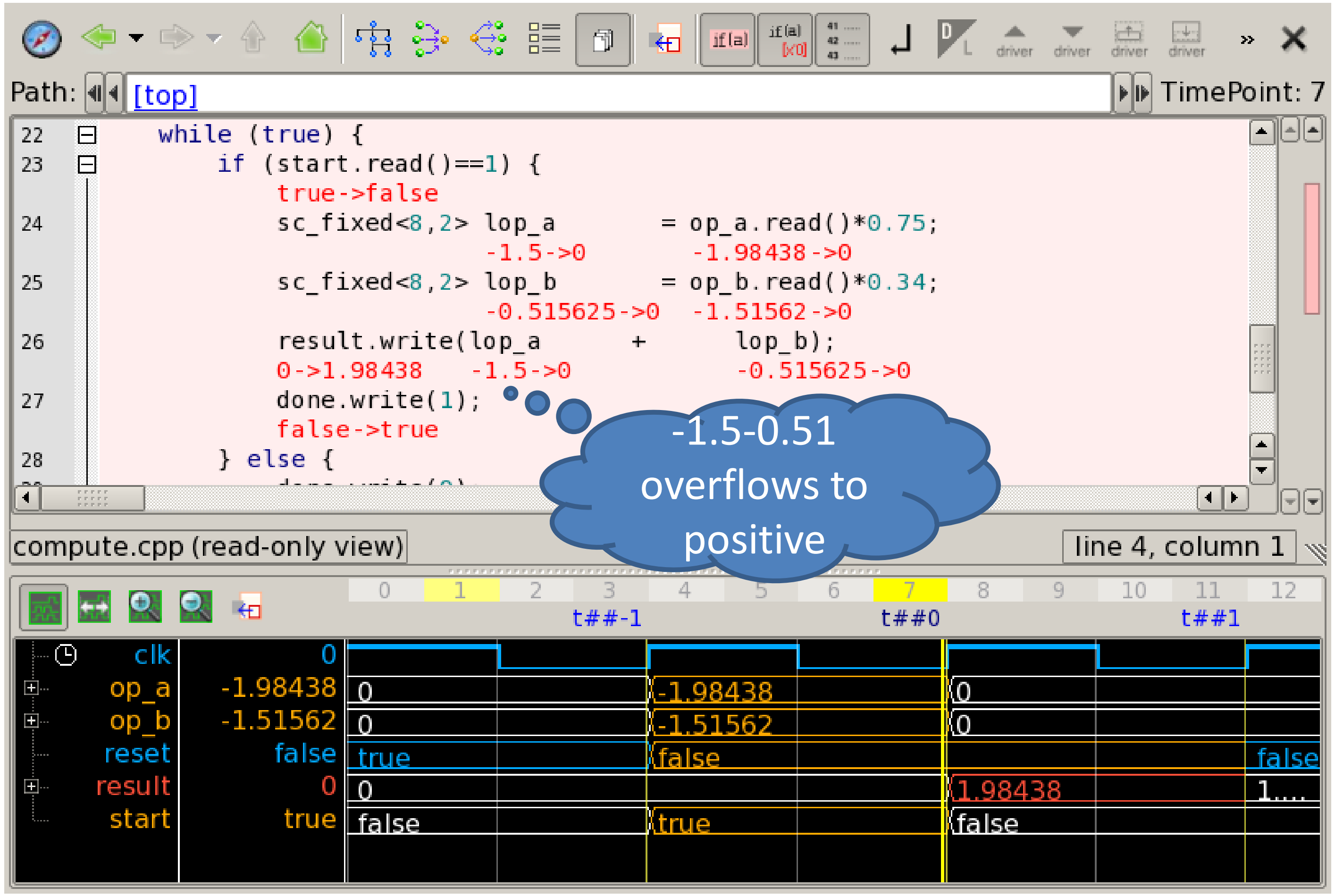


```
while (true) {
    if (start.read()==1) {
        result.write(op_a.read()*0.15+op_b.read()*0.34);
        done.write(1);
    } else {
```

```
-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
result[6]
mv>
```

1 redundant bit identified

Overflow Detection



```
    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 {
```

-1.5-0.51 overflows to positive

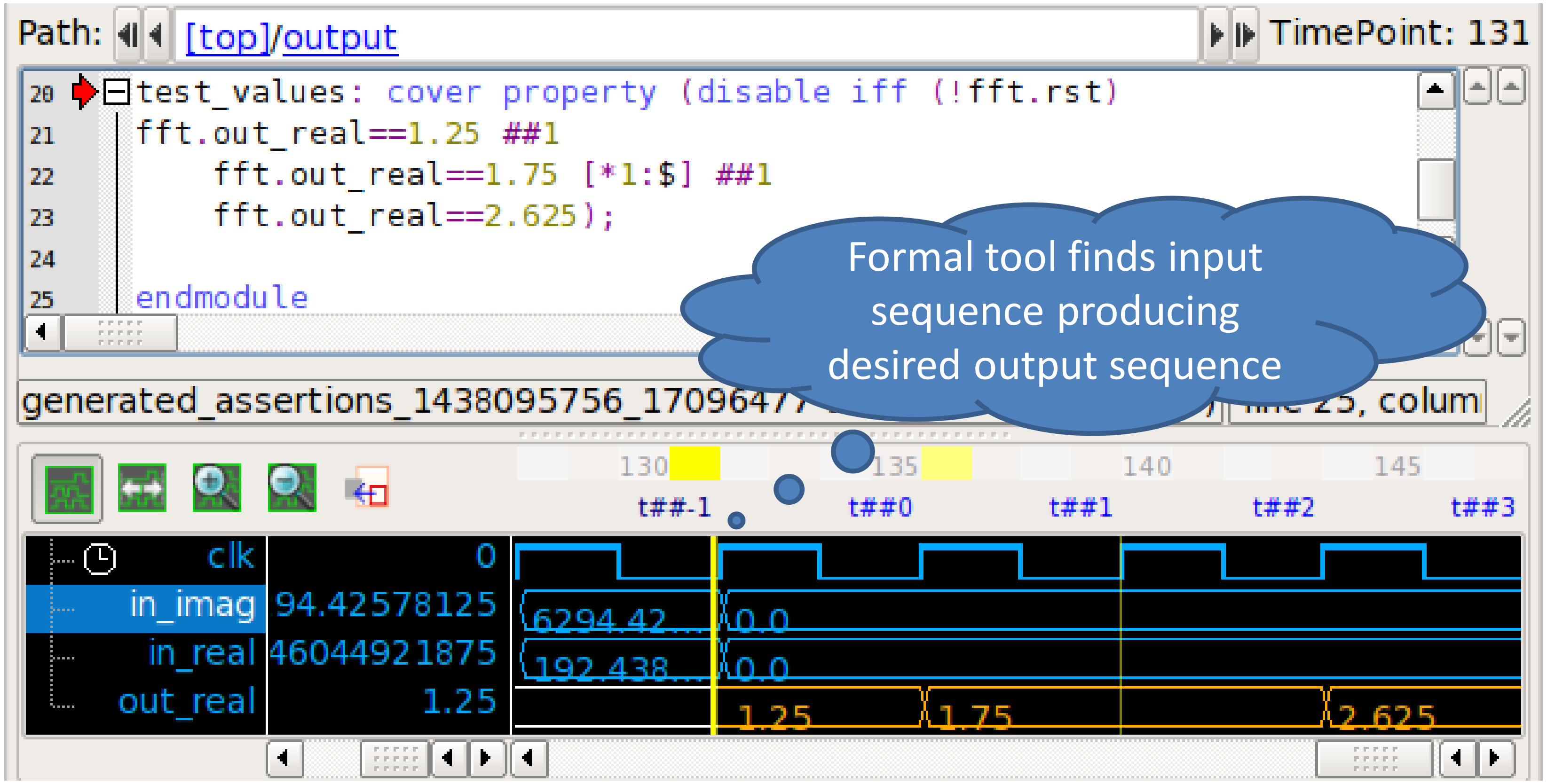
```
clk        0
op_a       -1.98438   0        -1.98438    0
op_b       -1.51562   0        -1.51562    0
reset      false      true     false                 false
result     0          0                    1.98438   1....
start      true       false    true        false
```

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



Path: [top]/output TimePoint: 131

```
test_values: cover property (disable iff (!fft.rst)
fft.out_real==1.25 ##1
    fft.out_real==1.75 [*1:$] ##1
    fft.out_real==2.625);

endmodule
```

generated_assertions_1438095756_17096477 ... line 25, colum

Signal	Value	
clk	0	
in_imag	94.42578125	6294.42... → 0.0
in_real	46044921875	192.438... → 0.0
out_real	1.25	1.25 → 1.75 → 2.625

130
135
140
145
t##-1
t##0
t##1
t##2
t##3
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?