



INTEL COMPILER FOR SYSTEMC

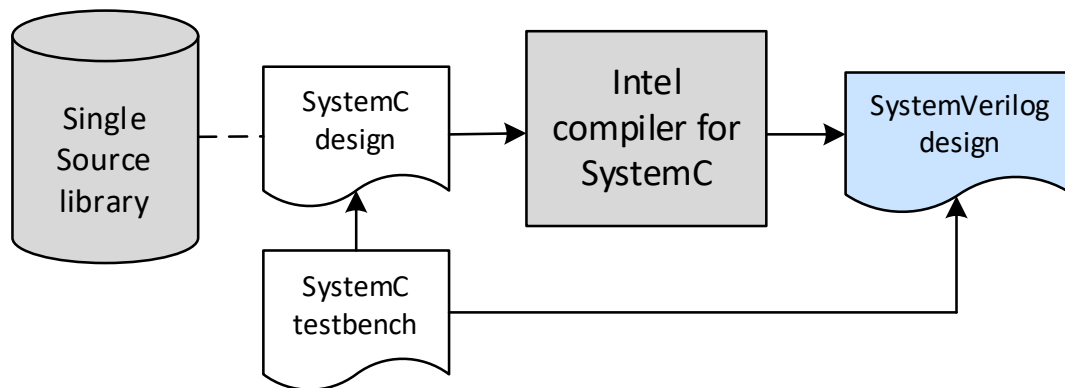
Mikhail Moiseev, Leo Azarenkov

Silicon and Systems Prototyping Lab, Intel Labs

December 2022

Introduction

- Intel® Compiler for SystemC* (ICSC) translates cycle accurate SystemC to synthesizable SystemVerilog



- Main goal is improving productivity of design and verification engineers
- Available under Apache License v2.0 with LLVM Exceptions
<https://github.com/chipsalliance/systemc-compiler>

**Other names and brands may be claimed as the property of others*

C++ and SystemC support

- ICSC uses SystemC 2.3.3, easy to switch to next
 - SystemC Synthesizable Standard supported except fixed/floating point
 - `sc_vector` supported
- Modern C++ standard support
 - C++11, C++14, C++17
 - Some STL containers: `std::vector`, `std::array`, `std::pair`
- Dynamic design elaboration, no limitations on elaboration stage
 - Arbitrary C++ in module constructors
 - Load input data from file/database
 - Enables to design highly reusable IPs

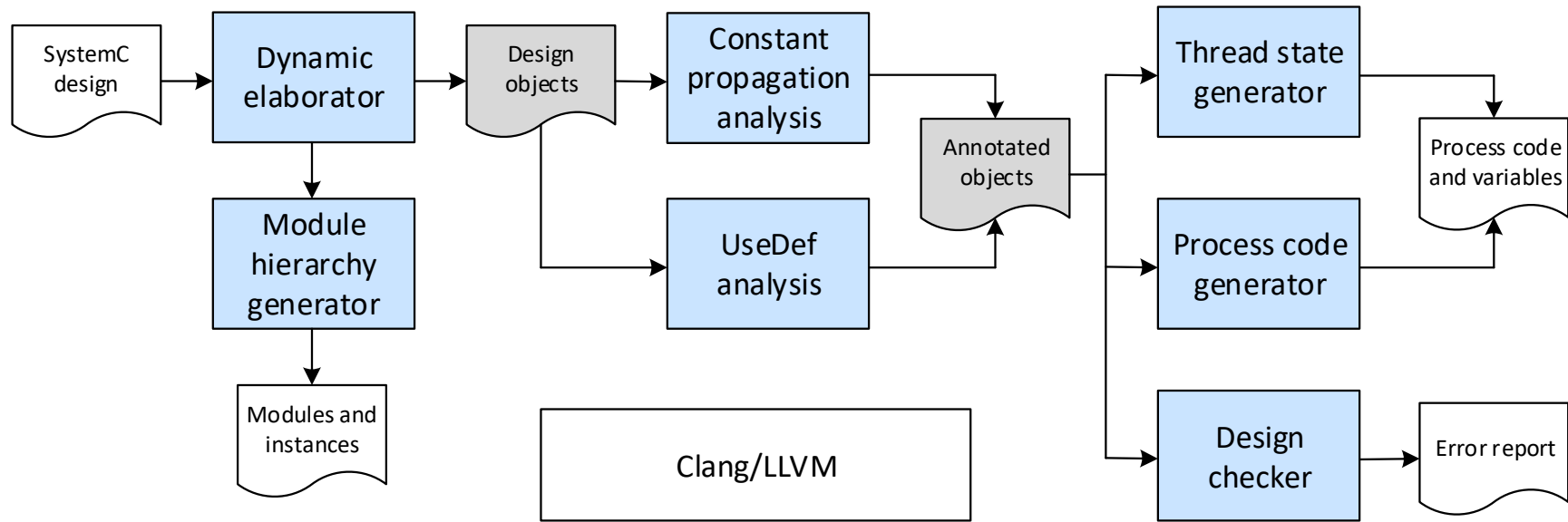
Fast and simple code translation

- ICSC does minimal optimizations, leaving others to logic synthesis tool
 - Constant propagation and dead code elimination
 - Used optimizations intended to generate better looking code
- ICSC works very fast
 - Elaboration takes several seconds
 - Code translation a few tens of seconds
- ICSC uses CMake build system
 - Provides CMake function runs the tool for specified top module and parameters
 - No build script or configuration files required
 - No tool specific pragmas

Human-readable generated Verilog

- ICSC generates SystemVerilog RTL which looks like SystemC source
 - Verilog variables have the same names everywhere it is possible
 - General structure of process/always block control flow is preserved
- Productivity advantages of human readable code
 - DRC and CDC bugs in generated Verilog can be quickly identified in input SystemC
 - Violated timing paths from ASIC logic synthesis tool can be easily mapped to input SystemC
- ECO fixes have little impact on generated Verilog

Tool architecture



Translation time results

Design name	Number of modules (instances)	Number of processes (instances)	Generated SV code, LoC	Translation time
A	58 (308)	161 (711)	29181	6 sec
B	19 (252)	65 (811)	20724	81 sec
C	78 (581)	291 (1470)	53404	18 sec
D	15 (57)	41 (146)	4662	2 sec
E	167 (880)	765 (2713)	87622	21 sec
F	53 (161)	173 (523)	25715	7 sec
G	57 (157)	170 (400)	21061	5 sec

Area and performance results ASIC

Design name	Intel Compiler for SystemC		Alternative implementation	
	Area	Freq MHz	Area	Freq MHz
A	4.2	761	3.9	752
B	26.7	694	23.4	686
C	91.5	400	81.2	377
D	169	763	158	769
E	20.9	769	19.6	771
F	70	1470	75	1420

Area and performance results FPGA

Design name	Intel Compiler for SystemC		Alternative implementation	
	Area Reg / LUT	Freq MHz	Area Reg / LUT	Freq MHz
A	2.4K / 10.2K	63	2.5K / 10.3K	62
B	54K / 145K	52	59K / 151K	53
C	15.7K / 46K	35	14.3K / 48K	25
D	547 / 1812	174	484 / 1823	172

SystemC assertion extension

- Immediate assertions
 - `SCT_ASSERT (RHS, EVENT)` – in module scope
- Temporal assertions
 - `SCT_ASSERT (LHS, TIME, RHS, EVENT)` – in module scope
 - `SCT_ASSERT_STABLE(LHS, TIME, RHS, EVENT);` – in module scope
 - `SCT_ASSERT_ROSE(LHS, TIME, RHS, EVENT);` – in module scope
 - `SCT_ASSERT_FELL(LHS, TIME, RHS, EVENT);` – in module scope
 - `SCT_ASSERT_LOOP (LHS, TIME, RHS, EVENT, ITER)` – in loop body in thread process
- ICSC translates immediate and temporal assertions into equivalent SVA

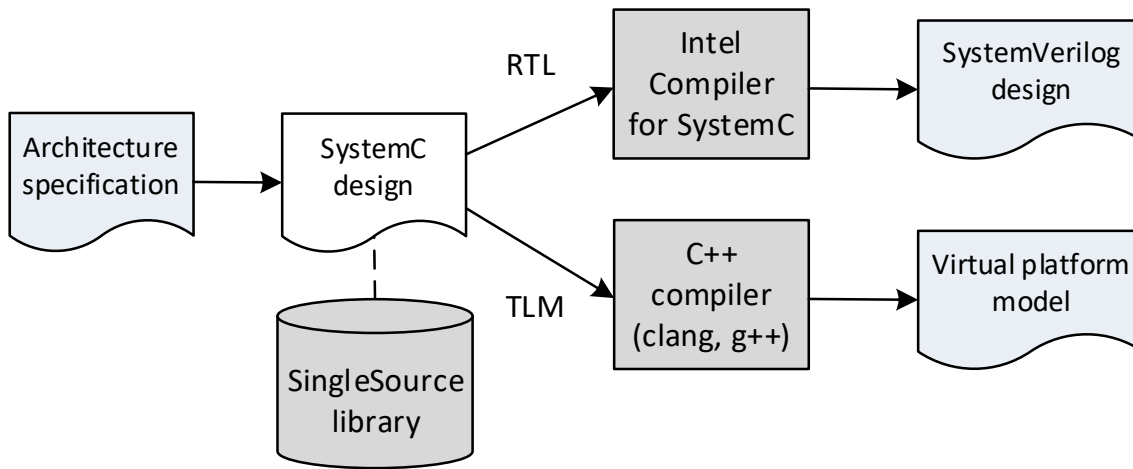
Assertion translation examples

```
static const unsigned N = 4;
sc_clk_in          clk{"clk"};
sc_in<bool>         req{ "req"};
sc_out<bool>        resp{"resp"};
sc_signal<sc_uint<8>> d{"d"};
sc_vector<sc_signal<bool>> e{"e",N};
...
// In module scope
SCT_ASSERT(req || !resp, clk.pos());
SCT_ASSERT(req, (1), resp, clk.neg());
SCT_ASSERT(req, (1,2), d.read()== N, clk);
SCT_ASSERT(e[0], (3), e[1], clk);
...
// In some process function
for (int i = 0; i < N; ++i) {
    SCT_ASSERT_LOOP(e[i], (2), e[i], clk, i);
}
```

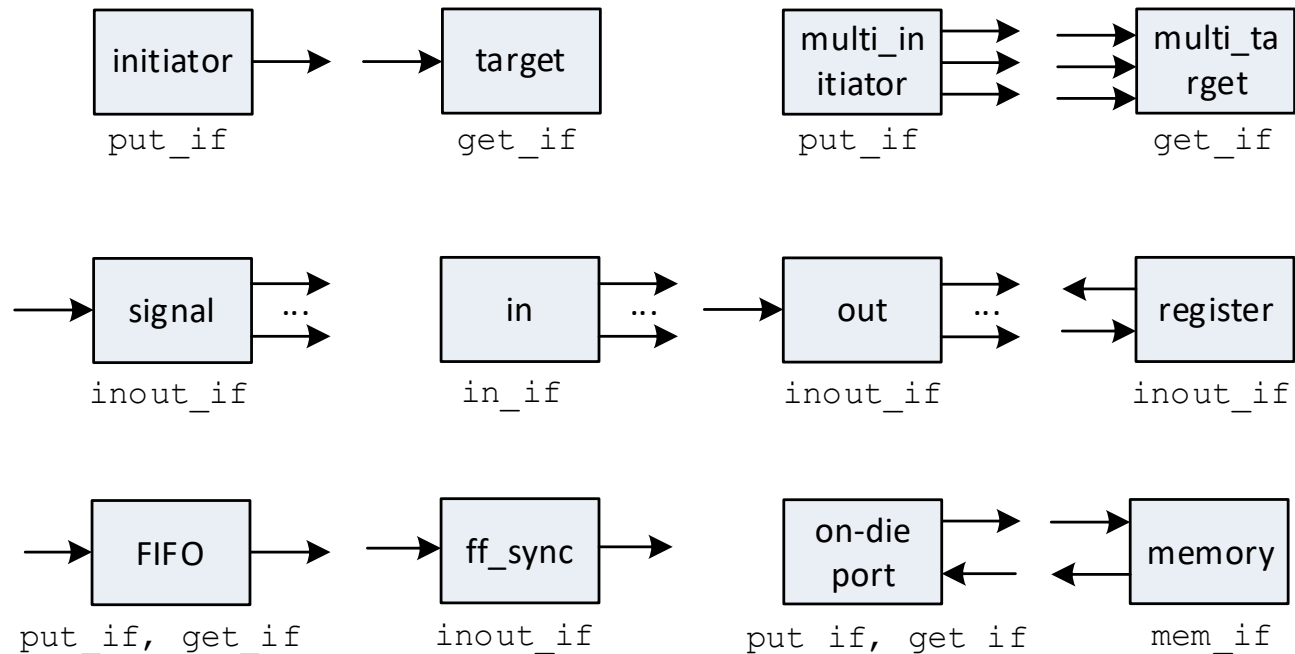
```
...
// In module scope
assert property(@(posedge clk) 1 |-> req || !resp);
assert property(@(negedge clk) req |>= resp);
assert property(@(clk) req |-> ##[1:2] d == 4);
assert property(@(clk) e[0] |-> ##3, e[1]);
...
// In some always block sensitive to clk
for (integer i = 0; i < 4; ++i) begin
    assert property (e[i] |-> ##2 e[i]);
end
```

Single Source Communication Library

- Single Source design flow uses the same SystemC code for
 - Digital design, SystemVerilog compatible with Intel ASIC/FPGA flow
 - Virtual platform, fast SystemC simulation model compatible with Simics

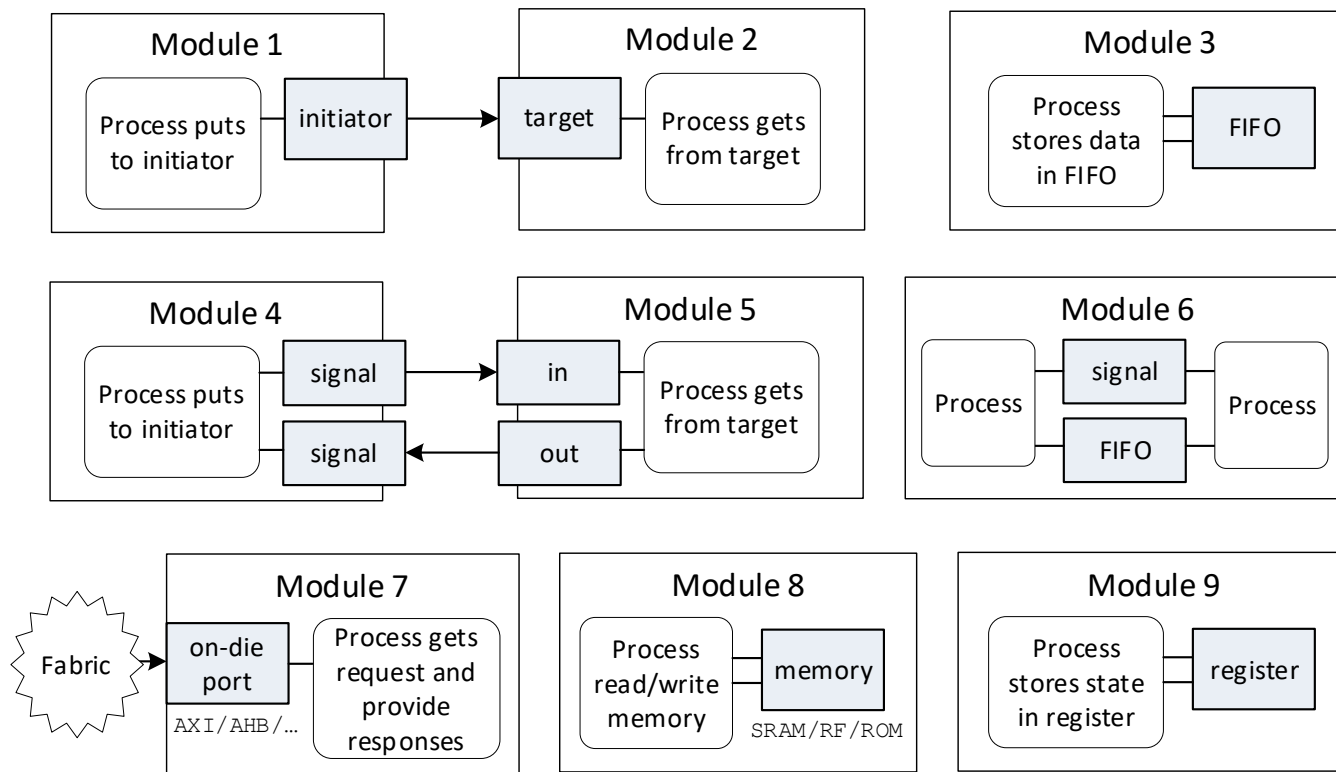


Single Source Library



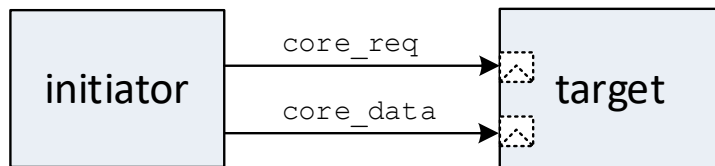
put_if, get_if, in_if, inout_if, mem_if – functional interfaces

Single Source use cases

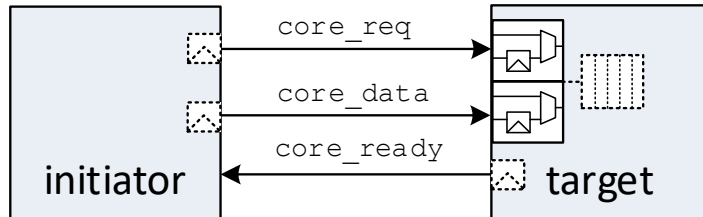


Initiator and target

Combinational connection



Buffered connection



- **Combinational connection**
 - For target which is always ready
 - Optional registers for request
 - Full throughput
- **Buffered connection**
 - Optional registers for request and ready
 - Optional FIFO
 - Protection from combinational loop
 - Full throughput in every mode

Initiator and target example

```
class A : sc_module {
    sct_initiator<T>      init{"init"};
    A(const sc_module_name& name){
        SC_THREAD(prodProc); sensitive << init;
    }
}

class B : sc_module {
    sct_target<T>         targ{"targ"};
    explicit B(const sc_module_name& name){
        SC_METHOD(consProc); sensitive << targ;
    }
}

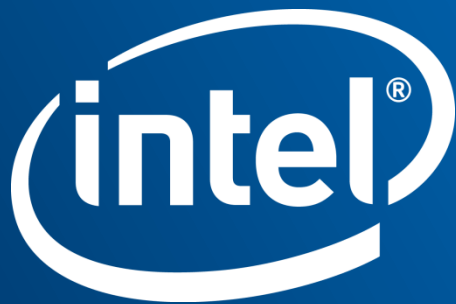
class Top : sc_module {
    A a{"a"}; B b{"b"};
    Top(const sc_module_name& name) {
        a.init.bind(b.targ); ...
    }
}
```

```
void prodProc() {
    init.reset_put();
    wait();
    while (true) {
        init.b_put(produceValue());
        wait();
    }
}

void consProc() {
    targ.reset_get();
    if (targ.request()) {
        consumeValue(targ.get());
    }
}
```


Conclusion

- Hardware design flow with ICSC differs from HLS tool flow
 - Lightweight source-to-source translation
 - Optimization works leaved for a logic synthesis tool
 - ICSC is used in multiple projects at Intel
- Efficient design and verification methodology
 - Immediate and temporal assertions synthesized to SVA
 - Increase abstraction level with SingleSource communication library
- Future plans
 - Migration to latest LLVM/Clang
 - Support of C++ classes as channel datatype
 - <https://github.com/chipsalliance/systemc-compiler/issues>



Agenda

- Intel® Compiler for SystemC
 - Introduction
 - Main features
 - Tool architecture
 - Code examples
 - Evaluation
- SystemC assertion extension
- SingleSource communication library
- Conclusion

Method process example

```
SC_CTOR(MyModule) {  
    SC_METHOD(methodProc);  
    sensitive << in << sig;  
}
```

```
void methodProc() {  
    bool b = in;  
    if (sig != 0) {  
        out = b;  
    } else {  
        out = 0;  
    }  
}
```

```
always_comb  
begin : methodProc    // test_process_simple.cpp:13:5  
    logic b;  
    b = in;  
    if (sig != 0)  
        begin  
            out = b;  
        end else begin  
            out = 0;  
        end  
end
```

Thread process example

```
CTOR(MyModule) {  
    SC_CTHREAD(thread1, clk.pos());  
    async_reset_signal_is(rst, false);  
}  
  
sc_in<unsigned> a{"a"};  
  
void thread1() {  
    unsigned i = 0;  
    while (true) {  
        wait();  
        unsigned b = i + 1;  
        i = i + a.read() + b;  
    }  
}
```

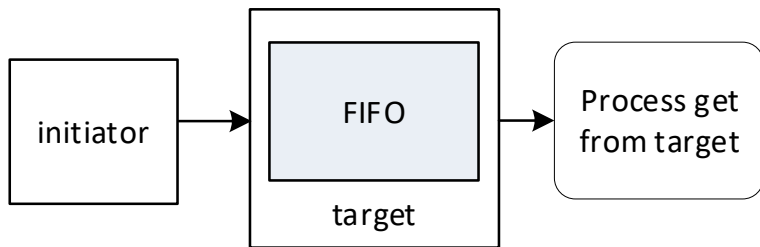
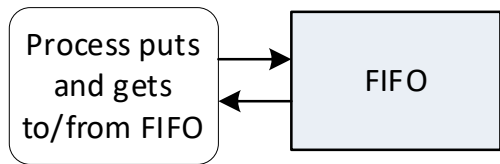
```
logic [31:0] a;  
logic [31:0] i, next_i; // Register variable  
always_comb begin  
    logic [31:0] b;      // Combinational variable  
    next_i = i;  
    b = next_i + 1;  
    next_i = next_i + a + b;  
end  
always_ff @(posedge clk or negedge rst) begin  
    if (~rst) begin  
        i <= 0;  
    end else begin  
        i <= next_i;  
    end  
end
```

Thread process with multiple states example

```
CTOR(MyModule) {  
    SC_CTHREAD(thread2, clk.pos());  
    async_reset_signal_is(rst, false);  
}  
  
void thread2() {  
    sc_uint<8> x = 0;  
    out.write(1);  
    wait(); // STATE 0  
    while (true) {  
        sc_uint<2> y = in.read();  
        x = y + 1;  
        wait(); // STATE 1  
        out.write(x);  
    }  
}
```

```
logic[2:0] x, x_next;  
logic PROC_STATE, PROC_STATE_next;  
always_comb simple_thread;  
function void simple_thread;  
    logic[1:0] y;  
    data_out_next = data_out;  
    x_next = x; PROC_STATE_next = PROC_STATE;  
    case (PROC_STATE)  
    0: begin  
        y = data_in; x_next = y + 1;  
        PROC_STATE_next = 1; return;  
    end  
    1: begin  
        out = x_next;  
        y = in; x_next = y + 1;  
        PROC_STATE_next = 1; return;  
    end  
    endcase  
endfunction
```

FIFO



■ FIFO usages

1. Inter-process communication inside of a module
2. Buffer for a process
3. Internal buffer for a target

■ FIFO parameters

- Size
- Synchronous/combinational for request and ready paths

FIFO example

```
sc_in<bool>          clk{"clk"};  
sc_in<bool>          nrst{"nrst"};  
sct_fifo<T, 2>       fifo{"fifo"};
```

```
explicit A(const sc_module_name& name) :  
sc_module(name) {  
    fifo.clk_nrst(clk, nrst);  
  
    SC_METHOD(producerProc);  
    sensitive << fifo.PUT;  
  
    SC_METHOD(consumerProc);  
    sensitive << fifo.GET;  
}
```

```
void producerProc(){  
    fifo.reset_put();  
    if (someCond && fifo.ready()) {  
        T val = getSomeVal();  
        fifo.put(val);  
    }  
}  
  
void consumerProc(){  
    fifo.reset_get();  
    T val;  
    if (fifo.get(val)) {  
        doSomething(val);  
    }  
}
```


What else

- Design correctness checking
 - Array out-of-bound, Dangling/null pointer dereference, Data races, ...
- Blackbox/memory insertion
 - Write SystemVerilog code in SystemC module or include *.sv/*.v file
 - SystemVerilog module name can be specified for SystemC module or module instance
 - No module generation option to use external implementation
- Method process with latches supported

Design template CMakeLists.txt

```
# Design template
project(mydesign)

# All synthesizable source files must be listed here (not in libraries)
add_executable(mydesign example.cpp)

# Source directory
target_include_directories(mydesign PUBLIC $ENV{ICSC_HOME}/examples/template)

# Add compilation options
# target_compile_definitions(mydesign PUBLIC -DMYOPTION)
# target_compile_options(mydesign PUBLIC -Wall)

# Add optional library, do not add SystemC library (it added by svc_target)
#target_link_libraries(mydesign sometestbenchlibrary)

# svc_target will create @mydesign_sctool executable that runs code generation
# ELAB_TOP parameter accepts hierarchical name of DUT
svc_target(mydesign ELAB_TOP tb.dut_inst)
```

How to install and run

- ICSC available under Apache License v2.0 with LLVM Exceptions
 - <https://github.com/intel/systemc-compiler>
- ICSC can be installed on Linux OS with C++17 compiler, cmake, git
- Install on Ubuntu 20.04 / 22.04 LTS
 - Clone the git repository <https://github.com/chipsalliance/systemc-compiler>
 - Set `ICSC_HOME` environment variable to the clone folder
 - Run install script which download and install LLVM/Clang/Protobuf/SystemC
- Run the tool
 - Run setup script
 - Create cmake target for my design, template design and examples provided
 - Run cmake and ctest

Thread process with loop example

```
void thread_loop() {  
    wait();                // STATE 0  
    while (true) {  
        for (int i = 0; i < 10; i++) {  
            k[i] = n[i] / m[i];  
            wait();        // STATE 1  
        }  
        wait();            // STATE 2  
    }  
}
```

```
function void thread_loop_func;  
    case (PROC_STATE)  
        0: begin  
            i_next = 0;  
            k[i_next] = n[i_next] / m[i_next];  
            PROC_STATE_next = 1; return;  
        end  
        1: begin  
            i_next++;  
            if (i_next < 10)  
                begin  
                    k[i_next] = n[i_next] / m[i_next];  
                    PROC_STATE_next = 1; return;  
                end  
            PROC_STATE_next = 2; return;  
        end  
        2: ...  
    endcase  
endfunction
```

Thread process with break example

```
void thread_break() {  
    wait();                // STATE 0  
    while (true) {  
        wait();            // STATE 1  
        while (!enabled) {  
            if (stop) break;  
            wait();        // STATE 2  
        }  
        ready = false;  
    }  
}
```

```
function void thread_break;  
    case (PROC_STATE)  
    0: begin  
        PROC_STATE = 1; return;  
    end  
    1: begin  
        if (!enabled) begin  
            if (stop) begin  
                // break begin  
                ready = 0;  
                PROC_STATE = 1; return;  
                // break end  
            end  
            PROC_STATE = 2; return;  
        end  
        ready = 0;  
        PROC_STATE = 1; return;  
    end  
    2: ...  
    endcase  
endfunction
```

Blackboxes

```
struct my_register : sc_module {  
    std::string __SC_TOOL_VERILOG_MOD__[] = R"(  
        module my_register (  
            input  logic [31:0] din,  
            output logic [31:0] dout  
        );  
        assign dout = din;  
        endmodule)";  
    SC_CTOR (my_register) {...} ...  
}
```

```
// VERILOG INTRINSIC  
module my_register (  
    input  logic [31:0] din,  
    output logic [31:0] dout  
);  
assign dout = din;  
endmodule
```

Method process with latches

```
#include "sct_assert.h"

void cgProc() {
    if (!clk_in) {
        enable = enable_in;
    }
    // To prevent error reporting for latch
    sct_assert_latch(enable);
}

void cgOutProc() {
    clk_out = enable && clk_in;
}

// Generated SystemVerilog
always_latch begin : cgProc
    if (!clk_in) begin
        enable <= enable_in;
    end
end
```