SystemC-to-Verilog Compiler:
a productivity-focused tool for hardware
design in cycle-accurate SystemC

Mikhail Moiseev, Intel Corporation
Roman Popov, Intel Corporation
Ilya Klotchkov, Intel Corporation
Introduction

• SystemC-to-Verilog Compiler (SVC) translates cycle-accurate SystemC to synthesizable Verilog code
• SVC is focused on improving productivity of design and verification engineers
  – Not a HLS tool
• SVC has multiple advantages which distinguish it from other tools
  – C++11/14/17 support
  – Arbitrary C++ at elaboration phase (in module constructors)
  – Fast and simple code translation procedure
  – Human-readable generated Verilog code
C++ and SystemC support

• SVC uses SystemC 2.3.3
  – SystemC Synthesizable Standard fully supported
• SVC supports modern C++ standards
  – C++11, C++14, C++17
  – Partial support of STL containers
• No limitations on elaboration stage programming, arbitrary C++ supported
  – Enables to design highly reusable IPs
  – Load input data from file/database
Fast and simple code translation

• SVC does minimal optimizations, leaving others to logic synthesis tool
  – Constant propagation and dead code elimination
  – Used optimizations intended to generate better looking code

• SVC works very fast
  – Elaboration takes several seconds
  – Code translation a few tens of seconds

• SVC uses conventional build system (CMake)
  – No build script or configuration files required
  – No code polluting with pragmas
Human-readable generated Verilog

• SVC generates Verilog 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 Design Compiler can be easily mapped to input SystemC
  – ECO fixes have little impact on generated Verilog
Tool architecture

- SystemC design
  - Dynamic elaborator
    - Module hierarchy generator
      - Modules and instances
    - Design objects
      - Constant propagation analysis
        - UseDef analysis
          - Annotated objects
            - Thread state generator
              - Process code generator
                - Design checker
                  - Error report
            - Process code and variables
          - Clang/LLVM
Design Elaboration

• SVC implemented as library on top of SystemC and Clang libraries
  – Substitutes SystemC library for linking
  – Runs Verilog code generation after SystemC elaboration phase
• SVC extracts design structure directly from process memory
  – Module hierarchy, links from pointers to pointee objects, values of scalar types, sensitivity and reset lists for processes
• SVC gets information about types form Clang AST
• Distinguish between pointer to dynamically allocated object and dangling pointer problem
  – Overriding \texttt{new} and \texttt{new[]} for \texttt{sc\_object} inheritors, and \texttt{sc\_new} for other types
Static Analysis and Code Generation

- Constant propagation analysis
  - Helps to determine number of loop iterations, eliminate dead code, substitute constant into generated code
- Used/defined variable analysis
  - Allows to split SystemC variables into local variables and registers
- Design correctness checking
  - Non-channel object read before initialization, Array out-of-bound access and dangling/null pointer dereference, Incomplete sensitivity lists for combinational methods, Inter-process communication through non-channel objects, ...
- Clocked thread state generation
- Clocked thread code generation
SC_METHOD example

```cpp
sc_in<bool>       in;
sc_out<bool>      out;
sc_signal<bool>   s;

SC_CTOR(MyModule) {
    SC_METHOD(method_proc);
    sensitive << in << s;
}

void method_proc () {
    bool a = in;
    if (s != 0) {
        out = a;
    } else {
        out = 0;
    }
}
```

```cpp
logic in, out;
logic s;

always_comb
begin // method_simple.cpp:112:5
    logic a;
    a = in;
    if (s != 0)
        begin
            out <= a;
        end else begin
            out <= 0;
        end
end
```
Clocked thread state generation

• SVC converts a thread into pair of `always_comb` and `always_ff` blocks
  – `always_ff` block implements reset and update logic for state registers
  – `always_comb` block contains combinational logic that computes the next state
• Clocked thread can have multiple states specified with `wait()`
  – Number of states is the number of `wait()` calls
  – Thread states are represented by automatically generated PROC_STATE variable
  – Main `case` of PROC_STATE in represents the SystemC thread FSM
• Thread variables divided into two groups
  – Local variables that are always assigned before use
  – Register variables that can retain their value from previous clock cycle
SC_CTHREAD example #1

sc_in<unsigned> a;
sc_out<unsigned> b;

SC_CTOR(MyModule) {
  SC_CTHREAD(thread1, clk.pos());
  async_reset_signal_is(rst, false);
}

void thread1 () {
n unsigned i = 0;
b = 0;
while (true) {
  wait();
b = i;
i = i + a;
}
}

logic [31:0] a;
logic [31:0] b, b_next;
logic [31:0] i, i_next;
always_comb begin // cthread_simple.cpp:101:5
  thread1_func;
end
function void thread1_func;
b_next = b; i_next = i;
b_next = i_next;
i_next = i_next + a;
endfunction
always_ff @(posedge clk or negedge rst)
begin : thread1_ff
  if (~rst) begin
    i <= 0; b <= 0;
  end else begin
    i <= i_next; b <= b_next;
  end
end
SC_CTHREAD example #2

```vhdl
sc_in<sc_uint<8>> a;
sk_out<sc_uint<8>> b;

SC_CTOR(MyModule) {
    SC_CTHREAD(thread2, clk.pos());
    async_reset_signal_is(rst, false);
}

void thread2() {
    sc_uint<8> i = 0;
    b = 0;
    wait(); // STATE 0
    while (true) {
        auto j = a.read();
        i = j + 1;
        wait(); // STATE 1
        b = i;
    }
}

logic PROC_STATE, PROC_STATE_next;
always_comb begin // cthread_simple.cpp:114:5
    thread2_func;
end
function void thread2_func;
    integer unsigned j;
    b_next = b; i_next = i;
    case (PROC_STATE)
    0: begin
        j = a; i_next = j + 1;
        PROC_STATE_next = 1; return;
    end
    1: begin
        b_next = i_next;
        j = a; i_next = j + 1;
        PROC_STATE_next = 1; return;
    end
endcase
endfunction
```
Clocked thread code generation

• The flow control statements *if*, *switch* and loops without *wait()* calls are converted into equivalent Verilog statements

• Loops with *wait()* calls are divided into several states
  – Loop statement in that case is replaced by *if* statement

• If loop with *wait()* contains *break* and *continue* statements they are replaced with the code up to the next *wait()* call
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_func;
    case (PROC_STATE)
        0: ...
        1: begin
            if (!enabled) begin
                if (stop) begin
                    // break begin
                    ready_next = 0;
                    PROC_STATE_next = 1; return;
                    // break end
                end
                PROC_STATE_next = 2; return;
            end
            ready_next = 0;
            PROC_STATE_next = 1; return;
        endcase
    endfunction
## Synthesis time for memory designs

<table>
<thead>
<tr>
<th>Design name</th>
<th>Number of modules (instances)</th>
<th>Number of processes (instances)</th>
<th>Generated code, LoC</th>
<th>Compilation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>58 (308)</td>
<td>161 (711)</td>
<td>29181</td>
<td>6 sec</td>
</tr>
<tr>
<td>B</td>
<td>19 (252)</td>
<td>65 (811)</td>
<td>20724</td>
<td>81 sec</td>
</tr>
<tr>
<td>C</td>
<td>78 (581)</td>
<td>291 (1470)</td>
<td>53404</td>
<td>18 sec</td>
</tr>
<tr>
<td>D</td>
<td>15 (57)</td>
<td>41 (146)</td>
<td>4662</td>
<td>2 sec</td>
</tr>
<tr>
<td>E</td>
<td>167 (880)</td>
<td>765 (2713)</td>
<td>87622</td>
<td>21 sec</td>
</tr>
<tr>
<td>F</td>
<td>53 (161)</td>
<td>173 (523)</td>
<td>25715</td>
<td>7 sec</td>
</tr>
<tr>
<td>G</td>
<td>57 (157)</td>
<td>170 (400)</td>
<td>21061</td>
<td>5 sec</td>
</tr>
</tbody>
</table>
## Area and performance results

<table>
<thead>
<tr>
<th>Design name</th>
<th><strong>SVC</strong></th>
<th></th>
<th><strong>Alternative implementation</strong>*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Area</td>
<td>Freq</td>
<td>Area</td>
</tr>
<tr>
<td></td>
<td>Reg / LUT</td>
<td>MHz</td>
<td>Reg / LUT</td>
</tr>
<tr>
<td>A</td>
<td>2.4K / 10.2K</td>
<td>63</td>
<td>2.5K / 10.3K</td>
</tr>
<tr>
<td>B</td>
<td>54K / 145K</td>
<td>52</td>
<td>59K / 151K</td>
</tr>
<tr>
<td>C</td>
<td>15.7K / 46K</td>
<td>35</td>
<td>14.3K / 48K</td>
</tr>
<tr>
<td>D</td>
<td>547 / 1812</td>
<td>174</td>
<td>484 / 1823</td>
</tr>
</tbody>
</table>

*Alternative implementation – Verilog code for these designs created in another way*
Future plans

• Temporal assertions in SystemC with automatic translation into SVA
• Cope problem with pointers (to avoid of new/new[] patch and sc_new)
  – Extend dynamic elaboration with static one
  – Preprocess files to replace new with sc_new
Questions