Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

SCCL: An open-source SystemC to RTL translator

Zhuanhao Wu*, Maya Gokhale†, Scott Lloyd‡ and Hiren Patel*
* University of Waterloo, Waterloo, Ontario, Canada
Email: {zhuanhao.wu,hiren.patel}@uwaterloo.ca
†Lawrence Livermore National Labs., Livermore, USA
Email: gokhale2@llnl.gov
‡ Brigham Young University, Provo, Utah, USA
Email: scott_lloyd@byu.edu

***Abstract*—We present SCCL, an open-source tool that translates SystemC designs into synthesizable register-transfer level (RTL). SCCL supports a subset of Accellera's SystemC synthesis standard based on the 2011 revision of C++. We use LLVM's Clang front-end to parse SystemC designs, and a suite of analysis passes to construct a SystemC-specific intermediate abstract syntax tree representation called Hcode. Hcode simplifies translation to other intermediate forms such as FIRRTL as well as direct transcription to SystemVerilog or VHDL. Currently, SCCL provides a translation phase to generate synthesizable SystemVerilog. Distinguishing aspects of SCCL include support for complex templated class descriptions that facilitate concise, parameterized hardware specification; introduction and full support for a new type of synthesizable channel called `sc_stream` that maps directly to standards such as AXI Stream, and a complete reference implementation targeting the Xilinx Vivado toolchain. We demonstrate SCCL's capabilities with a series of case studies including a highly templated SystemC implementation of the ZFP [1] floating-point codec. All case studies are deployed and executed on a Xilinx Zynq UltraScale+ FPGA platform.**

I. Introduction

SystemC is an electronic system-level design language embedded in C++ that offers the ability to quickly co-design complex hardware and software components to close the design productivity gap and meet time-to-market constraints. SystemC-based design methodologies are well established for the design of system-on-chips (SoCs) [2] as they permit the modelling of complex heterogeneous behaviours, fast simulation, and virtual prototyping. This is often achieved by using a mix of C++ code from libraries for drivers and tests, transaction-level models for fast simulation, and register-transfer level (RTL) designs for accurate representation of hardware components. SystemC encourages fast design-space exploration of an SoC.

Once the design-space exploration is complete, components chosen to be implemented in hardware are realized in one of the following ways: (1) rewriting the components as a single process amenable to high-level synthesis (HLS); (2) a complete rewrite using a hardware description language (HDL) such as SystemVerilog and VHDL or, (3) refining the SystemC design to meet Accellera's synthesis standard [3] and using SystemC synthesis tools from CAD or FPGA vendors. Approach (1) requires conflating a potentially multi-process description into a single process design. Approach (2) requires re-implementing the entire hardware component in the target language. Both of these approaches require careful verification to be done and result in significant lengthening of the design time. Approach (3) is the most natural path forward as Accellera's synthesizable standard is considerably large, but the availability of robust tools that synthesize complex SystemC designs is limited.

Commercial solutions such as Siemens' Catapult HLS, Cadence's Stratus HLS, and Xilinx's Vivado HLS provide support for SystemC. However, tools from CAD companies require a substantial monetary investment (O($100K USD) per year), which small-scale industries, research communities, and start-ups might find beyond reach. For FPGA targeting, Xilinx's Vitis has dropped support for SystemC.

We strongly believe that a free and community-supported SystemC synthesis or translator is needed to encourage a direct path from SystemC to synthesizable RTL. Consequently, we have developed SCCL, a Clang-based tool to translate SystemC designs to synthesizable RTL via an intermediate abstract-syntax tree (AST) called Hcode. SCCL supports complex C++ templated class descriptions, facilitating concise, highly parameterized hardware specifications. The translator recognizes a new type of synthesizable channel called `sc_stream` that maps directly to standards such as AXI Stream. The `sc_stream` channel bundles Ready/Valid/Data ports, simplifying interconnection of modules. SCCL also supports translation of both SystemC method and thread processes. For the latter, SCCL uses a novel thread process analysis algorithm. Note that SCCL is not an HLS tool and it does not perform optimizations that are common in modern HLS tools. This means any performance characteristics of the original design are retained in the generated RTL. SCCL's open-source repository [4] provides a complete reference implementation targeting the Xilinx Vivado toolchain. Our main contributions in this work are as follows:

- We present SCCL, an open-source Clang-based tool to translate SystemC [4] designs to synthesizable RTL. SCCL accepts a subset of Accellera's synthesis standard [3], and supports complex C++ constructs such as classes, class inheritance, templated types, user-defined classes, and virtual functions.
- We present an algorithm that translates SystemC thread processes to state machines. These state machines can be directly translated into RTL.
- We introduce a SystemC-specific intermediate AST,

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344, specifically LLNL LDRD 19-ERD-004.

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

Hcode, that simplifies translation to other lower-level IRs such as FIRRTL, and to SystemVerilog or VHDL.
- We explore SCCL's use with four case studies. The first case study is a moving average filter commonly used as a component in streaming applications. The second is a ZFP [1], [5] design that focuses on a novel floating-point compression technique important for high-performance scientific computing. The third is a systolic array design for matrix multiplication, which serves as one of the underlying components in many machine learning applications. The last case study is an integer divider using SystemC thread processes. The source code of SCCL and case studies can be accessed through [6].

II. Related work

Languages and frameworks to generate hardware has seen a resurgence in the recent past with the innovation of frameworks such as Chisel, PyMTL, and ROHD. In this section, we primarily concentrate our discussion on synthesis approaches that generate hardware from SystemC. Broadly, there are dynamic and static approaches to generating RTL from SystemC. Dynamic approaches execute the design by executing a portion of the SystemC simulation called elaboration, which concretizes design information such as module instances, their port bindings, and netlist. Static approaches analyze the SystemC source to extract the relevant design information. While dynamic approaches enable flexibility, static approaches are self-documenting as they do not depend on runtime information. Therefore, SCCL opts to use a static approach.

Similar efforts as ours include [7]–[9]. Authors of [7] present a bi-directional compiler between SystemC and Verilog. However, support for classes, class hierarchies, virtual functions, and template types appears to be missing. Authors in [8] present a tool to analyze SystemC designs, but have no support for its translation to gates. The most promising approach is Intel's SystemC-compiler [9] (ICSC), a dynamic approach, which does translate SystemC designs to synthesizable Verilog with support for several C++ constructs. ICSC's first public release was in late 2020, a couple of years into the development of SCCL, and due to ICSC's infancy at that time, we discovered that language support was not complete. ICSC has progressed further and a comparison with its translation would be considered in the future. Further, we want a path from SystemC to other target languages such as Chisel through FIRRTL, and potentially other intermediate representations such as MLIR, which, to our knowledge, are not planned for in ICSC. As a proof of concept of such a path, our tool targets an internal intermediate representation, Hcode, to balance the complexity between supporting SystemC features and amenability to synthesis, where other intermediate representations strive at the latter but lack at the former. We leave the extension to other intermediate representations as future works. As another aspect, SCCL includes support to synthesize an FPGA bitstream from the generated SystemVerilog and to run the generated design on a Xilinx Pynq board. SCCL's philosophy is to remain open-source and community-driven and supported.

TABLE I: SCCL Hcodes

Category	Description
SC object	module, port, signal, var, method, thread
types	typedefs for user-defined record types, with templates instantiated
bindings	port bindings and sensitivity lists
functions	module init and user-defined constructors, methods and functions

III. Translating SystemC to RTL

SCCL has three phases as shown in Figure 1: (1) parsing the SystemC design, (2) Hcode lowering, (3) and HDL generation. The first phase parses SystemC designs specified using Accellera's synthesizable 1.4.7 [3] standard and gathers structural information about the design. The second phase uses this structural information to transform Clang's AST to Hcode, an intermediate representation (IR) suitable for HDL generation. While Hcode is generic and could further target other IRs such as FIRRTL or MLIR, we provide a path to generate synthesizable SystemVerilog. In the third phase, we translate Hcode to SystemVerilog. To support a tangible hardware platform, the third phase includes tools to compile and deploy generated designs on an FPGA board.

A. Parsing and structure construction

SCCL uses Clang to parse and create an abstract syntax tree (AST) of the SystemC design [10]. SCCL traverses the AST using matchers to recognize and extract structural SystemC-specific definitions. This information includes instantiated SystemC modules, ports and signals of each module, process type (thread or method), member functions implementing the process, and so on. The structural representation delivers quick access to the underlying Clang AST and to the control-flow graph (CFG) of threads defined in the SystemC design.

B. Synthesizable channel type

Similar to the `sc_fifo` type in SystemC, a synthesizable channel type called `sc_stream` is introduced. It serves as a connector for data flows from one module to another. In conjunction with this channel type, two port types are also introduced, `sc_stream_in` and `sc_stream_out`. They serve as endpoints on modules for connecting channels. In contrast to the `sc_fifo` type, the `sc_stream` type and corresponding ports are synthesizable with an implementation that maps directly to an AXI Stream. Flow control on an `sc_stream` channel is realized by bundling Ready/Valid signals with the other data signals. Through the `sc_stream` template parameter, complex structures of signals can be specified for the data.

C. SystemC-specific translation to Hcode

The Hcode AST plays a central role in SCCL's design flow. The Hcode constructs describe SystemC instances and expresses the behaviour of the SystemC processes in a simplified AST. This simplified Hcode AST makes transcription to synthesizable RTL straightforward. Notice that Hcode AST is carefully designed to be be agnostic of the target language, and to allow transformations to other commonly-used IRs such as FIRRTL. Hcode opcode categories are shown in Table I. During

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

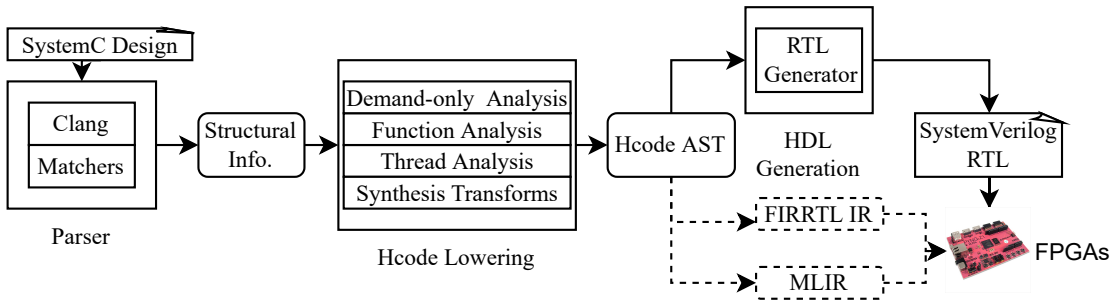


Fig. 1: Overview of SCCL's design flow.

the Hcode lowering phase, SCCL executes several analyses. We briefly describe a subset of these analyses next.

Demand-only analysis. We use a demand-driven analysis in generating the Hcode starting at the top-level SystemC module to any reachable submodules from there. SCCL has extensive support for templated SystemC modules. Consequently, Hcode lowering generates a unique definition for each template parameter combination encountered in templated module declarations. Shown in Listing 1 is a SystemC module with three template parameters with a definition for a version with `DIM=2`. The `hMethodCall` parameter (member function name) encodes the template instantiation: the module has been instantiated with `FP=<11,52>`, `B=64`, and `DIM=2`.

```
template<typename FP, typename B, int DIM>
  struct decode_stream;
template<typename FP, typename B>
  struct decode_stream<FP, B, 2>: sc_module
//...
hMethodCall zhw__decode_streamfp_t11_52_bits_t64_2...
              ...__get_window_func_1 [
      hVarref b_wrk_ms_proc_local_6 NOLIST
      hVarref dreg_bits_ms_proc_local_11 NOLIST
]
```

Listing 1: Templated module declaration and member function call.

The original SystemC design may have multiple templated types declared, but not necessarily used. Hcode only generates the templated types necessary for the design's implementation.

SCCL also supports the inheritance of SystemC modules, and user-defined C++ class declarations, and it resolves the use of overridden functions that can be statically determined. Within a SystemC module's scope, ports, signals, and variable declarations are instantiated, with templated data types resolved using the template parameter values provided by the structural information. We make considerable effort to preserve the names of declarations for readability of the generated RTL. Names of fields in structure types are also retained and field references are constructed to correlate to the original SystemC design.

Process analysis. Hcode lowering supports both SystemC methods and thread processes. We identify a subset of the supported features and challenges SCCL addresses.

C++ statements, local variables, SystemC constructs. Each process's body gets translated to Hcode. Many C++ statements such as conditionals, loops and function calls have corresponding constructs in RTL that are directly translated to Hcode AST. Examples of statements in this category include *if* and *switch* statements, looping statements such as *for* and *while*, and function calls. Most arithmetic operators translate directly to Hcode as well. However statements such as `break` require careful processing. A `break` in an `sc_method` translates directly to Hcode, but for a `sc_thread`, Hcode uses `return` within a function for semantically equivalent behavior.

Initialization of module-level variables and signals must be handled differently than those local to an `sc_process` (`sc_method` or `sc_thread`). At the module level, these objects are initialized in a reset block; in an `sc_process`, they are initialized at entry to the process. Variables declared in compound statements are promoted to the process level. Additionally for thread processes, all local variables are further promoted to module scope along with corresponding shadow registers to store state across clock boundaries.

Clang's AST includes classes and functions that are used to simulate the SystemC design, which we wish to ignore during Hcode generation. For example, the simulation of SystemC datatypes and the `sc_module` constructor functions are irrelevant to synthesis. SCCL recognizes such reference SystemC simulation constructs and discards the corresponding Clang AST's sub-tree. Member functions and operator calls related to SystemC types generate SystemC-specific Hcode, e.g. for signal and reads/writes and for operations on SystemC types such as `sc_int`. SystemC intrinsic functions are recognized and annotated in Hcode.

User-defined classes. In addition to using built-in SystemC types and functions, the design may define new, potentially templated, classes with associated constructors, operators, and methods. During Hcode generation, references to user-defined C++ classes and objects are resolved. Template parameters are instantiated. Type constructors and calls to template-type-resolved user-defined member functions used in a process are cataloged. Hcode for the member function calls is generated, and the functions invoked are added to a list of functions. In a SystemC module, all member functions referenced in any process are translated to Hcode. Function calls encountered in the traversal of a function body will also be added to the function catalog and their Hcode declarations generated. The module constructor is processed to translate port bindings and

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

Algorithm 1 runBuildFSM().

Require: SCFG G.
1: Let VB be a set of visited blocks.
2: Let VW be a set of visited blocks that are `wait`s.
3: Let WV be a list of CFG blocks that have `wait` statements.
4: Let $root$ be the root block of G.
5: buildFSM(root, VB, WV, VW)
6: **while** $WV \neq \emptyset$ **do**
7: $VB \leftarrow \emptyset$
8: $currBB \leftarrow WV.pop()$
9: buildFSM($currBB, VB, WV, VW$)
10: **end while**

Algorithm 2 buildFSM(bb, VB, WV, VW)

1: Let $toVisit$ be a list of CFG blocks to visit.
2: **if** bb has no successor **then return end if**
3: $VB \leftarrow VB \cup \{bb\}$
4: $toVisit.push(getSucc(bb))$
5: **do**
6: Let $currBB \leftarrow toVisit.pop()$
7: **if** $hasWait(currBB)$ and $currBB \notin VW$ **then**
8: $VW \leftarrow VW \cup \{currBB\}$
9: $WV \leftarrow WV \cup \{currBB\}$
10: **end if**
11: **if** ($isTwoSuccLoop(currBB)$ or $isCond(currBB)$) **then**
12: **do**
13: Let $loopVisited \leftarrow VB$
14: buildFSM($currBB, LoopVisited, WV, VW$)
15: **if** $hasUnvSucc(currBB) = false$ **then**
16: $VB \leftarrow VB \cup loopVisited$
17: **end if**
18: $currBB \leftarrow getUnvSucc(currBB)$
19: **while** $currBB$ is valid
20: **end if**
21: **if** $hasUnvSucc(currBB)$ and
22: $hasWait(currBB) = false$ **then**
23: $toVisit.push(getSucc(currBB))$
24: **else**
25: $toVisit.pop()$
26: **end if**
27: **while** $(toVisit.empty() = false)$

sensitivity lists to Hcode. In the case of arrays of modules or arrays of ports, port and sensitivity bindings may occur in fixed iteration *for* loops. Loops (potentially nested) are unrolled to generate individual bindings[1].

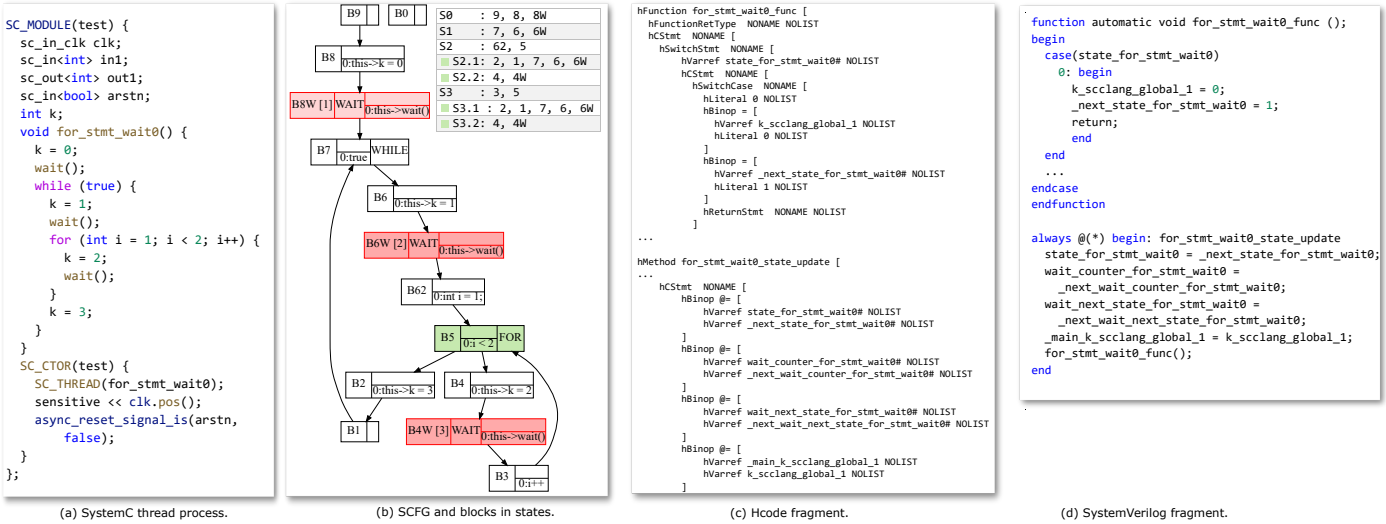
Thread analysis. SystemC processes come in two forms: methods and threads. The semantics of method processes are that of always blocks in SystemVerilog. Thread processes, however, are distinct to SystemC, which require careful translation into semantically equivalent RTL. A key distinguishing aspect about thread processes is that it encourages the use of `wait` statements. SystemC designs that use threads need to be converted into their corresponding state machines. Although this is not synthesis in the classical sense, the process of translating SystemC threads to state machines does result in assigning clock cycles to operations. SCCL uses a novel algorithm to translate SystemC threads into state machines amenable to synthesis. The algorithm has two steps. The first step converts the thread process's control-flow graph (CFG) into a suspension CFG graph (SCFG) that separates every CFG block with one or more `wait` statements such that every `wait` statement has its own CFG block. This is done while preserving predecessor and successor edges. The second step uses runBuildFSM and buildFSM from Algorithms 1 and 2, which uses a recursive algorithm mimicking a depth-first-search variant to traverse the SCFG and construct the CFG blocks that must execute in each corresponding state of the state machine. For brevity, we abstract the algorithm to focus on the traversal. We do not show the transformation from Clang's CFG to SCFG, and the collection of CFG blocks that provide the resulting RTL code.

The key intuition in the algorithm is that `wait`s denote the states, and the CFG blocks in between two reachable `wait` statements represent the behavior to execute in the corresponding state. Every occurrence of a `wait` changes the state; thus, the reachable paths between two `wait`s represent the behavior to execute. The algorithms implement this as follows. Algorithm 1 assumes the SCFG was constructed, and executes buildFSM to traverse the SCFG. The first execution of buildFSM starts at the $root$ block of the SCFG until the first occurrence of a `wait` statement. Since a `wait` denotes a state transition, one invocation of buildFSM ends when it encounters a `wait`, and defines the first state in the state machine (often the reset state). In addition, it is necessary to explore the reachable blocks from this encountered `wait`; hence, buildFSM records this in WV. This allows the algorithm to call buildFSM with the starting block for the traversal at the `wait` block inserted into WV. Notice that when each invocation of buildFSM starting at a `wait` backtracks and returns to runBuildFSM, a state in the state machine is defined. This continues until all `wait`s have been explored.

Once buildFSM is invoked in Algorithm 2, the algorithm uses a depth-first-search (DFS) approach with several unique features as explained next. (1) Whenever a `wait` is encountered, it stops the forward (deepening) traversal, records the `wait` block, and backtracks through the SCFG. Hence, `wait`s force the DFS to backtrack. (2) Conditional blocks and loop blocks with two successors are considered special blocks, and they must recursively invoke buildFSM while not affecting the blocks visited (VB). The main idea is to use buildFSM to explore the subgraph starting at these special blocks. However, we explore these subgraphs without affecting the traversal that caused the DFS to arrive at these blocks. Notice that this is essential because traversals may repeatedly visit blocks that were already visited in the traversal so far, which standard DFS does not permit. This is important to support because a traversal from a special block is perceived as a new traversal. Consequently, we invoke buildFSM, but using $loopVisited$ to ensure VB does not get updated during the recursive invocation. It does, however, get updated after there are no successors. For a block that is neither a `wait` nor the special blocks, Algorithm 2 retrieves the successor, if one exists, and ensures that it can be visited next via inserting it into $toVisit$. This repeats until there are no blocks to visit.

[1]A limited form of subscript expression is currently supported.

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.



(a) SystemC thread process. (b) SCFG and blocks in states. (c) Hcode fragment. (d) SystemVerilog fragment.

Fig. 2: Illustrative example of analyzing SystemC thread process.

Example. Figure 2 highlights the main features of the algorithm. The SystemC thread process has three `wait`s with the code behavior simply assigning values to a variable (Figure 2(a)). Figure 2b shows the SCFG generated by SCCL. Notice that `wait`s have their own block while preserving the predecessor and successor relationships. The table on the right shows the blocks traversed at every invocation of buildFSM. The first invocation of buildFSM starts at block 9 until it reaches the `wait` in block 8W. Then buildFSM backtracks resulting in the path taken to produce the first state as shown in the table by S0. Block 8W was inserted into WV to start the next traversal at this block and explore the SCFG to find the next reachable `wait`. The next invocation of buildFSM starts at block 8W and traverses into the while loop until it reaches block 6W. Once again, the block is inserted into WV, and the traversal ends by backtracking up to the starting block. This results in state S1 and its corresponding visited blocks. So far, this example illustrates the terminating condition for the traversal to include `wait`s, one of the first novelties in the algorithm. Starting at block 6W, the traversal encounters a special block at block 5. This is because it is a loop that has two successors ($isTwoSuccLoop$): one where the loop condition evaluates to true (block 2) and the other to false (block 4). The algorithm recursively calls buildFSM while preserving the history of visited blocks VB. Thus, the algorithm traverses through the true path of the `for` loop through the `while` loop and back to block 6W where it begins backtracking to the starting block 5. Since there is another successor, the path of visited blocks is recorded as S2.1, and the traversal continues on the false path ending at block 4W yielding S2.2. Since block 6W is a `wait` and it has already been visited as recorded in VW, block 6W is not a candidate for the starting block of the traversal again. We only need to traverse from the starting block 6W once. However, block 4W has not served as the starting block for the traversal. Therefore, the next traversal starts at block 4W, revisits block 5, and once again traverses through the true and false paths as shown in S3.1 and S3.2. Note that `break` statements within loops are seamlessly supported with buildFSM. This is because a `break` statement results in an edge to a block outside of the scope of the loop, which the current traversal approach accommodates. Since buildFSM has used all `wait` blocks as starting blocks, all states of the state machine have been generated as shown in the table.

Hcode thread transforms. The generated Hcode for a thread process has three components: a process body `hFunction` containing the executable statements in the `sc_thread`, a `hMethod` containing the state update control logic, and an initiator `hMethod` to initialize state control variables and call the body `hFunction`. Figure 2(c) shows that the Hcode generates a `switch` statement in the body `hFunction` that selects the blocks to be executed according to the current state as shown in the table in Figure 2(b). `wait`s get transformed into state variable updates within a `case` in the body `hFunction`. The `hMethod` updates a collection of control variables including current and next state, and counters to count down `wait` states for `wait`s that require more than one clock cycle of waiting.

Hcode thread generation. The generated RTL (Figure 2(d)) mirrors the three components in the Hcode: a synchronous process that updates states and tracks *persistent signals* whose values must persist across `wait` statements; a function that determines next states and persistent signal values in the next clock cycle; and, a combinational process that drives the persistent signals in the next clock cycle. These three components correspond to the state update `hMethod`, `hFunction`, and the initiator `hMethod`, respectively. The synchronous process implements the signal and state changes *across* `wait` statements and is clocked. For example, the value of `k` is persistent across `wait` statements, which corresponds to `_main_k_scclang_global_1`. Every clock cycle, `_main_k_scclang_global_1` is updated with `k_scclang_global_1`, where the latter is an auxiliary

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

variable that tracks to the last value before a `wait` is called in the SystemC design. The synchronous process captures the reset logic specified by `async_reset_signal_is()` in SystemC. The function implements the signal and state changes *between* `wait` statements with a `switch` statement. Consider the two `wait` statements on lines 12 and 15, which corresponds to S2. Depending on whether the `for` loop is executed or not, `k_scclang_bloal_1` is either updated with 2 on line 14 or with 3 on line 17. Finally, the combinational logic ensures the initialization of the auxiliary variables and calls the function.

D. RTL generation

SCCL generates HDL from Hcode AST employing Python. Currently, SCCL supports the generation of synthesizable SystemVerilog. SCCL parses Hcode AST with the lark library [11] to get a recursive representation of the AST. SCCL then transforms the Hcode AST representation through multiple passes. Separating the translation into multiple passes facilitates future extensions. We highlight two passes in the HDL generation process, in the order they take place.

Type collection and expansion (TCE). The TCE pass first identifies and stores the Hcodes of user-defined types by traversing the AST. This is possible because the Hcode AST maintains the types and names of members of each C++ class in an `hTypedef` Hcode. Next, TCE traverses all variable declarations with a user-defined type, including port definitions, variables, and signals and flattens these declarations into individual members, with the stored `hTypedef` Hcode, where the member names are concatenated after the original variable name. TCE also supports variables declared as an array, in which case a variable of user defined types will be expanded into arrays of members. Each generated array, representing a single member, has the same dimension as the original array. Consider a C++ class with two integer member fields: `class A {public: int a; int b;}`, and a variable declared as `A arr[2]`. TCE expands `arr` into two arrays expressed in SystemVerilog as `logic[31:0] arr_a[0:1];` and `logic[31:0] arr_b[0:1];`.

Assignment specification (AS). AS pass determines whether an assignment in the SystemC design should be translated into a blocking assignment (BA) or a non-blocking assignment (NBA) in a procedure, and is specific to SystemVerilog. In SystemVerilog, a BA from `1` to a variable `a` takes the form of `a = 1` while a NBA takes the form of `a <= 1`, where both of them can be written as `a = 1` in SystemC. Briefly, the difference between a BA and a NBA is that accessing a variable assigned by BA immediately will obtain the assigned value while accessing a variable assigned by NBA will obtain the value assigned by the last execution of the procedure. The correct use of BA and NBA is crucial to maintain the correct behavior of the design. By default, the semantics of a SystemC assignment is BA. SystemC models the delayed assignment by using specific types such as `sc_signal`. The AS pass identifies variables declared as `sc_signal`s and translates the assignments to these variables as NBA, while assignments to other variables such as local variables and C++ primitive types are translated into BA. The AS pass may result in a mix of BA and NBA, which is avoided in manually written SystemVerilog since the mix may lead to unexpected results by the designer. This mix of BA and NBA is required to maintain the SystemC and C++ behavior and is managed automatically by SCCL. We believe that the mixing of BA and NBA is benign, and that it is important to preserve as much as possible the original description of the design in the generated RTL.

E. CI Integration

SCCL's development uses continuous integration (CI) to ensure that changes and additions by the developers and the community are automatically integrated into the code repository, and tested for correctness. Since SCCL supports translating SystemC design to Hcode and Hcode to SystemVerilog, we need facilities to provide software testing, and hardware testing. We perform the software testing via doctest [12] and hardware testing using pytest [13]. Both of these are driven by CMake's ctest, and executed by the repository's automated actions whenever a commit is merged into the main repository.

Docker Instance. We use a Docker instance with all the pre-requisites such as Clang, SystemC, and other build tools deployed in the instance to accelerate the compile and test execution with the CI. For each version of Clang, we publish and maintain the corresponding Docker instance. An additional benefit is that we can use the Docker instance for development, which eliminates the need to modify the developer's underlying system.

CI for Parser and Hcode. doctest provides a fast and convenient unit testing framework for SCCL's parser and Hcode generation. Currently, SCCL uses doctest for three types of tests: (1) tests for specific data structures for its implementation, (2) tests for its newly-introduced algorithms such as the state machine generation from SystemC threads, and (3) tests to ensure correct parsing of structural information. (1) allows us to ensure the correct operation of data structures used in SCCL, and they also serve as examples of how to use these data structures. (2) ensures that the new algorithms we introduce in SCCL are verified. For example, the algorithm to translate SystemC threads into state machines for synthesis was carefully verified with manually crafted tests and several tests included with SystemC-Compiler. Since the development of SCCL promotes plugins, such tests do not require including the plugin that generates the Hcode. This allows one to focus on the plugin responsible for generating the state machine for SystemC threads, and expedite debugging. (3) confirms that the structural information extracted from SystemC designs has correctly been parsed. These are all enabled with doctest, and their integration with ctest. A user can select any subset of tests to execute during development.

CI for RTL generation. SCCL uses pytest for testing the generation of RTL. There are two types of RTL generation tests: (1) tests for RTL generation from Hcode, and (2) tests for SCCL's ability to generate RTL in an end-to-end fashion,

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

converting SystemC designs into SystemVerilog RTL design. (1) ensures SCCL correctly performs passes on Hcode, such as TCE and AS, and generates correct RTL code. For example, these tests ensure that user-defined types are expanded into their expected field names. (2) ensures the overall functionality of SCCL. Each test case in (2) converts a SystemC design into its corresponding RTL design, and consists of the following three steps enabled by the `pytest-steps` plugin. First, the test attempts to generate the Hcode from the SystemC design from the parser. Second, the test attempts to generate RTL design from the Hcode specification. The generated Hcode and RTL design can be checked against a golden Hcode and RTL design that is known to be correct. Finally, the test executes the CAD tool (Vivado) given the generated RTL design as an input to ensure the design is synthesizable. A subset of RTL generation tests can be selected to execute during development similar to the CI for parser and Hcode.

IV. Empirical evaluation

We demonstrate SCCL with four case studies: (1) a moving average filter, (2) a hardware implementation of a floating-point codec in a format customized to scientific data called ZFP [5], (3) a systolic array design for matrix multiplication, and (4) an integer divider case study using SystemC thread processes. We use Xilinx's Vitis toolchain to compile the generated RTL to target the Ultra96-V2 board's Xilinx UltraScale+ FPGA. Note that Xilinx's latest toolchains, Vivado, Vitis HLS, and Vitis only support using SystemC designs as analytical models and do not support the synthesis of RTL design in SystemC. These case studies show SCCL's ability to accept complex SystemC and C++ constructs and generate synthesizable hardware. The synthesized hardware is deployed, executed, and validated on the FPGA platform.

```
template<int BW>
SC_MODULE(divider) {
  typedef sc_uint<BW> val_t;
  typedef sc_bigint<2 * BW> tmp_t;
  ...
  sc_in<val_t> dividend;
  sc_in<val_t> divisor;
  sc_in<bool>  valid;

  sc_out<val_t>  quotient;
  sc_out<bool>  vld;

  ...
  for(int i = BW - 1; i >= 0; i--) {
    if(_temp[i + 1].read() + _op2[i + 1].read() <= _op1[i + 1].read()) {
      _temp[i] = _temp[i + 1].read() + _op2[i + 1].read();
      _quotient[i] = _quotient[i + 1].read() | (one << i);
    } else {
      _temp[i] = _temp[i + 1].read();
      _quotient[i] = _quotient[i + 1].read();
    }
    _op2[i].write(_op2[i + 1].read() >> 1);
    _op1[i].write(_op1[i + 1].read());
    _vld[i].write(_vld[i + 1].read());
  }
  ...
};
```

Listing 2: Division calculation with templated operand size.

A. Moving average calculation

```
template<int WINDOW_SIZE>
SC_MODULE(moving_average)
{ ...
  sc_signal <data_t> window[WINDOW_SIZE];
  sc_signal<sc_uint<8>> n, insert;
  sc_signal<data_t> sum;
  sc_signal<data_t> cur_min, cur_max, cur_avg;
  sc_signal<bool> datardy;
  sc_signal<data_t> _cur_min, _cur_max;
  sc_signal<data_t> dividend;
  sc_signal<data_t> divisor;
  sc_signal<bool> div_vld;
  sc_signal<bool> div_out_vld;
  ...
  divider<DATAW> u_div { "u_div" };
  ...
  dividend.write((sum.read().to_uint() + datastrm.
    data.read().to_int() ));
  divisor.write((n.read().to_uint()+1));
  div_vld.write(datastrm.valid_r());

  if (datastrm.valid_r()) { // new data
    if (_cur_min > datastrm.data) _cur_min = datastrm
    .data;
    if (_cur_max < datastrm.data) _cur_max = datastrm
    .data;
    window[insert.read().to_uint()] = datastrm.data;
    if (n.read().to_uint() < WINDOW_SIZE) n.write(n.
    read().to_uint() +1);

    sum.write(sum.read().to_uint() + datastrm.data.
    read().to_uint() - window[insert.read().to_uint()
    ].read().to_uint());
    if ((int) insert.read() >= WINDOW_SIZE-1) insert.
    write(0);
    else insert.write(insert.read() + 1);
    datardy = true;
  }
  ...
}
```

Listing 3: Moving average calculation with templated window size.

A common building block in various filtering operations is computing the moving average of a stream of values. This case study presents SystemC designs that compute such moving averages. Specifically, we present three designs each differing in the optimizations they offer for the division operation, which is central to the computation. The first design, *MA-Div*, is the simplest where the SystemC design uses the division operator (e.g. `a/b`) as shown in Listing 3. This results in a direct mapping of the division operator to SystemVerilog. Naturally, this yields a complex combinational division logic that challenges timing closure for frequencies beyond 15Mhz on the Ultra96-V2 board. Although the FPGA platform provides DSP slices, they were not inferred by the FPGA synthesis tools. As our first optimization, we changed the division operation to a bitwise shift, resulting in our second design, *MA-Shift*. Although this limits the division to be in multiples of two, the synthesized design offers an improvement over the first, allowing the design to reach the target frequency of 100Mhz. The second SystemC design is not general. To address this, we implement a third SystemC design, *MA-Div-Sub*, that fully utilizes the hardware resources available in the DSP slices by implementing a fully pipelined divider as shown in Listing 2. The divider takes as

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

Configurations	*MA-Div* LUT	Reg.	*MA-Shift* LUT	Reg.	DSP	*MA-Div-Sub* LUT	Reg.	DSP
DW=16, WS=16	333	381	117	344	3	407	846	5
DW=16, WS=64	581	1151	383	1108	3	690	1646	5
DW=64, WS=16	1348	1485	467	1448	5	3592	9175	7
DW=64, WS=64	2012	4561	1296	4516	5	4656	12375	7

TABLE II: Resource utilization of moving average with different parameterization. DW is short for DATAW; WS is short for WINDOW_SIZE.

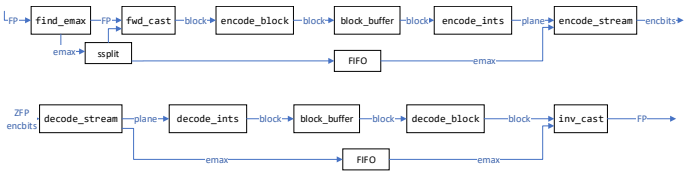


Fig. 3: ZFP hardware with encoder (top) and decoder (bottom) [16].

input the streams of dividend and divisor and performs *division by subtraction algorithm* [14] to calculate the quotient. The divider is parameterized by a template parameter that denotes the width of the dividend and divisor. Notice that the divider is a submodule u_div within the moving average module, where the running average value of a window of an input data sample stream is generated as an output stream.

The window size is a template parameter to the moving average module as shown in Listing 3. Note the use of the SystemC read and write methods to get and update the sc_signal class variables. The signals' template parameter is itself a templated class sc_uint<DATAW> where DATAW is a compile-time integer constant. In this example, it has been set to 64.

Table II shows the resource utilization of the moving average modules. The *MA-Div* modules reach a frequency of 15Mhz and do not utilize any DSP slices on the FPGA. With *MA-Shift*, we are able to achieve a frequency of *100Mhz*, while utilizing DSP slices. Moreover, *MA-Shift* requires fewer LUT and registers compared to *MA-Div* as the division is replaced with shifting logic. *MA-Div-Sub* modules achieve a frequency of 100 MHz, utilizing more DSP slices compared to *MA-Shift*. Table II shows the results for the moving average designs with DATAW set to 16 or 64, and with WINDOW_SIZE set to 16 or 64. The results show SCCL's capability of handling template parameters such as WINDOW_SIZE, and compile-time constants such as DATAW.

B. ZFP: Floating point codec

ZFP [5] is a well established floating-point codec with both software [15] and hardware [16] implementations. ZFP is intended for use with scientific multi-dimensional arrays that describe physical phenomena. We use SCCL to map a synthesizable SystemC implementation of the ZFP algorithm to SystemVerilog, and then onto the FPGA.

The high level flow of the ZFP hardware encoder and decoder is shown in Figure 3. The ZFP hardware algorithm is highly templated, as is shown in a small code fragment Listing 1. The top level encoder and decoder designs have template parameters floating-point format FP, array dimension D and streaming data width W used. Template parameters enable flexibility in the implementation. FP is itself a user-defined class with parameterized exponent and fraction sizes. In the ZFP hardware algorithm, user-defined classes that include these templated types are used in SystemC ports and signals.

This complex design, with 1800 lines of SystemC, compiles automatically through SCCL to RTL, passes through the Vivado 2022.1 tool chain, and runs correctly on the Ultra96-v2 board.

Table III demonstrates the resource utilization for the ZFP hardware encoder and decoder. The encoder and the decoder are of type zhw::encode<FP, D, W> and zhw::decode<FP, D , W> in C++ respectively. For both the encoder and decoder, FP takes fp_t<E, F>, which is a user-defined template type that denotes a floating-point format with E-bit exponent and F-bit fraction. Our case study uses 32-bit (fp_t<8,23>) and 64-bit (fp_t<11,52>) floating-point format that comply with the IEEE-754 standard. We demonstrate an encoder configuration compressing 64-bit floating-point numbers and outputting a 64-bit data stream (zhw::encode<fp_t<11,52>,2,64>, E1), and a lightweight encoder compressing 32-bit floating-point numbers and outputting a 32-bit data stream (zhw::encode<fp_t<8,23>,2,32>, E2). Both E1 and E2 achieve a clock frequency of 50MHz on the FPGA. We also present two decoder configurations, one decompressing 64-bit data stream to get 64-bit floating-point numbers (zhw::decode<fp_t<11,52>,2,64>, D1), and another decompressing 32-bit data stream to get 32-bit floating-point numbers (zhw::decode<fp_t<8,23>,2,32>, D2). The decoder configurations D1 and D2 mirror the encoder configurations with a clock frequency of 20MHz on the FPGA. The reported frequencies represent the characteristics of the ZFP hardware data path which are preserved by SCCL.

Table III shows the resource utilization for the two encoder configurations and two decoder configurations (E1, E2, D1, and D2) with different hardware parameters. The 64-bit encoder (E1) takes up the most resources on the FPGA with 12262 LUTs, 16969 registers, and other resources. The encoder that compresses 32-bit floating-point numbers, E2, takes up significantly fewer resources, consuming 2525 LUTs, 1153 registers, and other resources. The reduction in resource utilization compared to its 64-bit counterparts arises from smaller structures for holding internal data and a reduced level of encoding logic. We observe a similar trend for D1 and D2. The decoder decompressing 64-bit data stream, D1, consumes 14380 LUTs and 19974 registers, while D2, operating on a 32-bit data stream, only consumes 3135 LUTs and 6780 registers. Note that the changes between different configurations are achieved by solely varying the template parameters. Hence, we show SCCL's value in rapid prototyping, leveraging templates in C++.

C. Systolic array

Systolic array (SA) architectures for matrix multiplication (MMULT) are essential in many of today's machine learning applications. Examples of SA for MMULT are already employed in various commercial accelerators such as Google

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

Configurations	E1		E2		D1		D2	
Module Name	LUT	Reg.	LUT	Reg.	LUT	Reg.	LUT	Reg.
Total	12262	16959	2525	1153	14380	19974	3135	6780
block_buffer	582	2057	68	212	1030	2052	5	4
(en/de)code_block	7151	11221	482	696	5129	13241	2179	5156
(en/de)code_ints	1421	87	1115	79	5387	1076	180	46
(en/de)code_stream	1553	239	817	126	969	478	105	38
find_emax	859	2114	25	21				
(fwd/inv)_cast	589	1029	5	5	1335	1036	402	513
(ssplit_ex/cast_buffer)	122	212	17	14	514	2037	257	1013
Lines of code		1779		1823		1728		1728

TABLE III: ZFP hardware codec resource utilization on Ultra96-V2 with different parameterization. Lines of code of the generated SystemVerilog are reported.

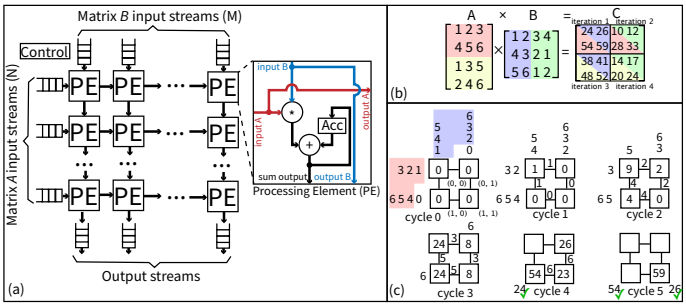


Fig. 4: (a) $N \times M$ systolic array. (b) Example partition of matrix for performing MMULT on a 2×2 SA. (c) Iteration of SA.

TPU [17] and Xilinx DPU [18].In this case study, we show SCCL's ability to expedite the design of a SA for MMULT. Our design is parameterized by the size of the SA and the type of operands the SA operates on, specified by C++ template parameters. A SA consists of regularly arranged processing elements (PEs) programmed to perform common operations such as multiplications and additions, and the PEs communicate with each other through streaming interfaces. Figure 4a shows an example of our SA design with N rows and M columns of PEs that perform MMULT for two matrices A and B. Input matrices are streamed into the SA, and both input and output are buffered using FIFOs.

The process of MMULT by SA is split into multiple iterations, where each iteration computes a distinct sub-matrix of size $N \times M$ in the product matrix. Hence, the size of the SA is agnostic of the input matrices. MMULT partitions matrices A and B into sub-matrices of N rows and M columns, respectively. Distinct pairs consisting of blocks from A and B are fed into the SA to produce the $N \times M$ product (sub-)matrix. For example, consider multiplying a 4×3 matrix A and a 3×4 matrix B using a 2×2 SA. Matrix A is partitioned into two 2×3 sub-matrices and B is partitioned into two 3×2 matrices as shown in Figure 4b. Then, the 2×2 SA array goes through 4 iterations, each taking a pair of sub-matrices from A and B, and produces 2×2 sub-matrices in the resultant matrix C.

Each PE performs the dot product of a row from A and a column from B, where the underlying operation is the multiply-accumulate (MAC) operation. Each PE in our SA has two input ports and three output ports on its data path. The PE also uses a register (Acc) for storing accumulated values. At every clock cycle, the PE takes the input from A and B, multiplies them, and accumulates the product in Acc. Meanwhile, inputs A and B are forwarded to outputs A and B so that a neighboring PE can consume the values. When a PE finishes its computation, the results are sent through the sum output port to a neighboring PE so that it can be streamed out of the SA.

Configurations	8×8		16×16		8×8-RTL		16×16-RTL	
Module Name	LUT	Reg.	LUT	Reg.	LUT	Reg.	LUT	Reg.
Total	6410	4771	24961	18139	6362	4770	24765	18139
pe	5418	3944	22751	16336	5464	3944	22945	16336
control_inst	4	4	5	5	5	6	5	8

TABLE IV: SA resource utilization on Ultra96-V2 with different parameterization.

Figure 4c illustrates one iteration of running the MMULT on a 2×2 SA. Initially, at cycle 0, the data values of the first 2×3 and 3×2 blocks from A and B respectively are fetched and buffered for the systolic array. Also, all accumulators are initialized to zeros. Note that padding zeros are inserted by the control logic to guarantee correct results. At cycle 1, the PE at $(0, 0)$ takes both input values, 1 and 1, and performs MAC with the accumulator, updating the accumulator with 1. Note that these inputs are passed to the neighboring PEs. At cycle 2, the PE at $(0, 0)$ takes the input values, 2 and 4, performs MAC with the accumulator and obtains 9 $(2 \times 4 + 1)$. The PE at $(0, 1)$ takes the input values passed from the PE at $(0, 0)$ and the input buffer, which are 1 and 2, and performs MAC, resulting in 2. The process continues and at cycle 3, the result of PE at $(0, 0)$ is ready (24) and can be sent to the sum output port. At cycle 4, the dot product result of PE at $(0, 0)$ reaches the output stream buffer and is ready to be fetched. Similarly, the results of PE at $(0, 1)$ and $(1, 0)$ reach the output stream buffer and can be fetched at cycle 5.

Figure IV shows the 8×8 and 16×16 systolic array mapped onto the Ultra96-V2 FPGA board with a target frequency of 100Mhz. We additionally show the same designs implemented manually using SystemVerilog, labeled as 8×8-RTL and 16×16-RTL. In both SCCL generated designs and manually implemented designs, the PEs take up the most resources. The 8×8 design takes up a total of 6410 LUTs and 4771 registers, while the 16×16 design takes up a total of 24961 LUTs and 18139 registers. This matches the fact that the design scales linearly with respect to *the number of PEs*. Similarly, 8×8-RTL design takes up a total of 6362 LUTs and 4770 registers, while the 16×16-RTL design takes up a total of 24765 LUTs and 18139 registers. The differences in resource utilization between the SCCL generated designs and manually implemented designs are negligible. Hence, when generating RTL designs, SCCL does not incur additional hardware costs.

D. Design using threads

We illustrate SCCL's support for synthesizing SystemC thread processes using an integer divider example. This case study is similar to the divider in the moving average except that the divider is using SystemC threads. Listing 4 shows the division-by-subtraction algorithm implemented with SystemC

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

```
SC_MODULE(divider) {
  typedef sc_uint<BW> val_t;
  typedef sc_biguint<2 * BW> tmp_t;
  sc_in<bool> clk;
  sc_in<bool> arst;

  sc_stream_in<val_t> dividend;
  sc_stream_in<val_t> divisor;
  sc_stream_out<val_t>  quotient;

  void thread_proc_1() {
    _vld.write(false);
    _op1_vld.write(false);
    _op2_vld.write(false);
    wait();

    while(true) {
      if(_sync.read()) {
        _op1.write(dividend.data_r());
        _op2.write(tmp_t(divisor.data_r()));
        _op1_vld.write(true);
        _op2_vld.write(true);
        wait();

        _op1_vld.write(false);
        _op2_vld.write(false);
        wait();

        // perform div-sub division
        _quotient = 0;
        _temp = 0;
        _op1 = dividend.read();
        _op2.write(tmp_t(divisor.read()));
        _vld.write(false);
        wait();

        for(int i = BW - 1; i >= 0; i--) {
          if(_temp.read() + (_op2.read() << i) <=
    _op1.read()) {
            _temp.write(_temp.read() + (_op2.read()
    << 1));
            _quotient.write(_quotient.read() | (1LL
    << i));
          }
          wait();
        }
        _vld.write(true);
      }
      if(_sync_out.read()) {
        _vld.write(false):
      }
      wait();
    }
  }
  ...
  }
```

Listing 4: Division-by-subtraction using threads.

threads. The design accepts a dividend stream and a divisor stream as input. When values are valid in both streams, the design starts calculating the quotient, which is sent to an output stream. The design is parameterized by the number of bits in the dividend and divisor, `BW`. In the divider design in Listing 4, the use of SystemC threads enables a sequential description of the division-by-subtraction algorithm. Note that the divider design splits the division into multiple steps and is not pipelined, as pipelining is not the semantics of threads in SystemC. SCCL translates this design with SystemC threads into a SystemVerilog design using combinational and sequential hardware as described in the thread analysis section.

Configurations	LUT	Reg.	DSP
`BW=8`	86	68	4
`BW=16`	165	100	4
`BW=32`	357	164	6
`BW=64`	695	292	9

TABLE V: Resource utilization of divider.

Table V shows the resource utilization of the synthesized design with the `BW` of both dividend and divisor being 8, 16, 32, and 64. In all configurations, the design can be mapped onto proper FPGA resources, such as LUTs, registers, and DSP slices. For example, for `BW=64` configuration, the design utilizes 695 LUTs, 292 registers, and 9 DSP slices. The resource utilization shows SCCL's capability of mapping a SystemC thread onto FPGA with adequate resources. All configurations achieve an operating frequency of 100 MHz.

V. Conclusion

SCCL provides an open-source translator from SystemC to synthesizable RTL capable of handling complex C++ constructs such as templated types, class hierarchies, inheritance, and virtual functions. We also present an approach to translate SystemC thread processes into synthesizable RTL. To allow translation to other IRs and frameworks, we develop an intermediate AST called Hcode, with a reference implementation translating Hcode to SystemVerilog. We illustrate SCCL's use with four case studies including various moving average calculation SystemC designs, a ZFP floating point codec, a systolic array design, and an integer divider unit. Note that the ZFP case study makes extensive use of advanced C++ constructs and template parameterization. SCCL generates the RTL for each of these case studies, and we deploy it on an FPGA. We envision continued community and industry support in making SCCL a strong, and free alternative to commercial solutions.

References

[1] L. Noordsij, S. v. d. Vlugt, M. A. Bamakhrama, Z. Al-Ars, and P. Lindstrom, "Parallelization of variable rate decompression through metadata," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020, pp. 245–252.

[2] M. Goli and R. Drechsler, "Automated design understanding of systemc-based virtual prototypes: Data extraction, analysis and visualization," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 188–193.

[3] "Accellera's SystemC Synthesis 1.4.7," https://www.accellera.org/downloads/standards/systemc, 2016.

[4] "SCCL: An open-source tool that translates SystemC to RTL." [Online]. Available: https://github.com/anikau31/systemc-clang

[5] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, pp. 2674–2683, Dec. 2014.

[6] Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: FCCM Artifact Evaluation," 2023. [Online]. Available: https://zenodo.org/record/7813660

[7] C. Huang, H. Gao, Y. Zhong, and S. Cai, "A high-performance bidirectional compiler for conversion between systemc and verilog," in *Proceedings of the 6th International Conference on High Performance Compilation, Computing and Communications*, ser. HP3C '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 124–130. [Online]. Available: https://doi.org/10.1145/3546000.3546019

Z. Wu, M. Gokhale, S. Lloyd, and H. Patel, "SCCL: An open-source SystemC to RTL translator," in proceedings of International Symposium On Field-Programmable Custom Computing Machines (FCCM), May 2023, pp. 1–10.

[8] A. Kaushik and H. D. Patel, "Systemc-clang: An open-source framework for analyzing mixed-abstraction SystemC models," in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, 2013, pp. 1–8.
[9] M. Moiseev, R. Popov, and I. Klotchkov, "SystemC-to-Verilog Compiler: a productivity-focused tool for hardware design in cycle-accurate SystemC," in *Proceedings of DVCON Europe*, 2020.
[10] "LLVM/Clang Framework," https://clang.llvm.org/.
[11] "Lark - a parsing toolkit for Python," https://github.com/lark-parser/lark.
[12] "doctest: C++ Testing Framework," https://github.com/doctest/doctest.
[13] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, "pytest 5.0," 2004. [Online]. Available: https://github.com/pytest-dev/pytest
[14] G. Hawkes, "DSP: Designing for Optimal Results," *High-Performance DSP Using Virtex-4 FPGAs, Xilinx*, 2005.
[15] "ZFP," https://github.com/LLNL/zfp.
[16] "ZHW - ZFP Hardware Implementation," https://github.com/LLNL/zhw.
[17] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246
[18] Xilinx, "Vitis AI User Guide (UG1414)," https://docs.xilinx.com/r/1.2-English/ug1414-vitis-ai/Deep-Learning-Processor-Unit-DPU.