EasyChair Preprint

№ XXX

Single Source library for high-level modelling and hardware synthesis

Mikhail Moiseev and Nanda Kalavai

October 20, 2024

2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Single Source library for high-level modelling and hardware synthesis

Mikhail Moiseev, Intel, Hillsboro, USA (*mikhail.moiseev@intel.com*)
Nanda Kalavai, Intel, Folsom, USA (*nanda.g.kumar.kalavai@intel.com*)

***Abstract*— In this paper we present a design methodology that utilizes a single source code for high-level modelling and hardware synthesis. The methodology is based on Single Source library developed in SystemC that contains high-level communication channels to provide fast simulation and cycle accurate modes transparently for the design.**

Keywords—hardware modelling;hardware design;SystemC; high level synthesis

I. INTRODUCTION

Digital hardware design includes multiple steps where several artifacts are developed. Before logic synthesis, a typical hardware design flow usually has architecture exploration, functional and performance modelling, hardware IP RTL design, SoC design, verification, virtual prototyping, and software/firmware development. Hardware modelling is performed iteratively with fixing and improving models based on simulation results and bugs found. In IP RTL development and SoC design, which are often done in parallel with hardware modelling. Some functional, performance and architectural issues can be discovered that causes updates to the architecture specification. Having multiple independent models, which are developed in parallel and often by different teams, leads to the problems such as: proving equivalence between the models and RTL, sharing updates between the models and RTL, extra resources for multiple models' development and verification.

We propose a design methodology which implements high-level models and hardware synthesis from a single source code, see Figure 1. Our design methodology is based on SystemC language as it meets requirements of performance models, IP RTL design, SoC design and verification. SystemC gives multiple advantages over conventional hardware description languages, such as high flexibility and parametrization, full power of underlying C++ language, STL and other libraries. To support features of high-level models and RTL synthesis in the same SystemC code, we have developed Single Source library which consists of high-level communication channels and process macros. These channels operate in two modes: *Fast simulation mode* and *Cycle accurate mode*.

Fast simulation mode is intended for all kinds of modelling done prior to RTL design. In this mode, a SystemC design with the Single Source channels is compiled with a C++ compiler like GCC or Clang. In Fast simulation mode there are no clock events, the clocked thread processes are activated by events of the communication channels listed in their sensitivity lists. That provides simulation speed up in comparison with simulation of conventional

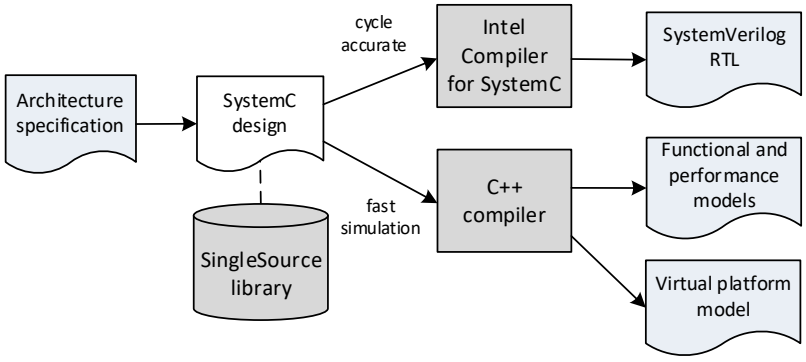

Figure 1. Hardware design with Single Source library

SystemC designs. *Cycle accurate mode* is intended for high-level synthesis. In this mode the Single Source channels have implementation which complies to SystemC synthesizable standard [1]. For hardware synthesis we are using Intel Compiler for SystemC [2] which is an open-source tool under Apache 2.0 license available at [3].

The key advantage of the proposed methodology is that Single Source library can be used by hardware architects to model the architecture intent and micro architects as well as RTL designers can seamlessly update the architecture model to RTL. This unified approach helps to reduce time to market since RTL is an updated version of architecture model rather than RTL development starting from scratch after all the functional and performance models are tuned.

II. SINGLE SOURCE LIBRARY

Single Source library includes interfaces, macros, communication channels and auxiliary classes. The library is open source under Apache 2.0 license at [3]. The main use cases of the communication channels are presented in Figure 2. These channels can be used in SystemC method and thread processes created with SC_METHOD and SC_THREAD correspondently. A SystemC process accesses a channel should have the channel in the process sensitivity list.

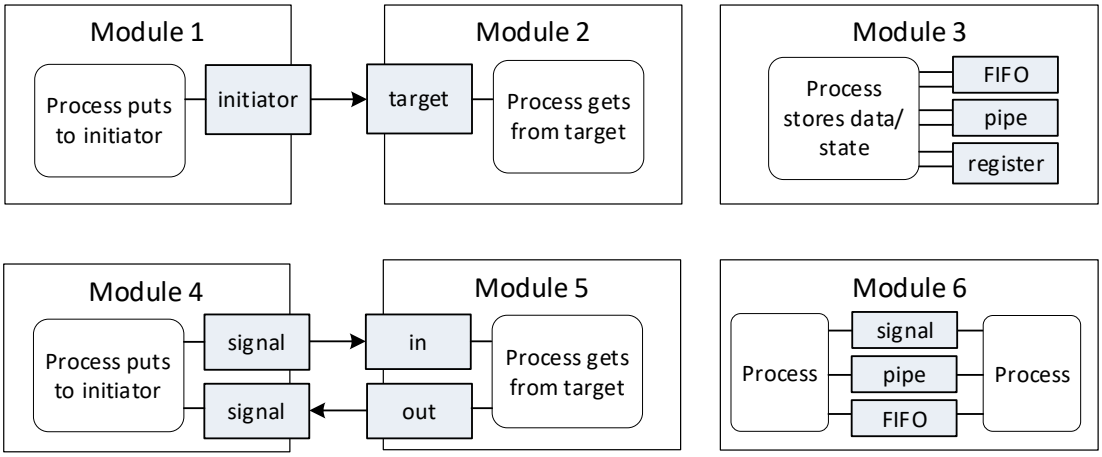

Figure 2. High-level communication channels

A. *Single Source Interfaces*

The Single Source channels provide functional interfaces with may-blocking and non-blocking functions, some of them listed in Table I.

B. *Initiator and Target*

Initiator and Target are channels intended to connect two processes in different modules: one process puts data into Initiator, other process consumes the data from the corresponding Target. Initiator implements *sct_put_if* interface, Target implements *sct_get_if* interface. Initiator and Target can provide combinational or buffered connections and can have a pipeline register in the request path (see Figure 3). The response path contains an implicitly added register if that is required to avoid combinational loop. Buffered connection can have an optional FIFO. Combinational connection can be used if the target process is always ready to consume, otherwise buffered connection should be used. Both connections provide maximal throughput. More details are given at [4].

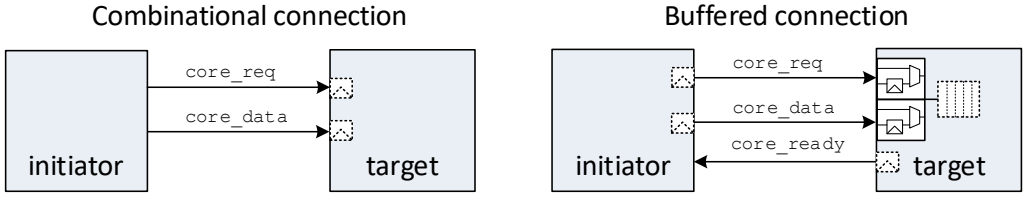

Figure 3. Initiator and Target connections

Table I. Single Source interfaces

Interface	Function	Comment
sct_put_if	bool ready()	Return true if channel is ready to put request
	void reset_put()	Reset this channel
	bool put(const T& data)	Non-blocking put request into channel if it is ready, return ready to request
	void b_put(const T& data)	May-blocking put request, could be used in thread process only
sct_get_if	bool request()	Return true if channel has request to get
	void reset_get()	Reset this channel
	T peek()	Peek request, return current request data, if no request last data returned
	T get()	Non-blocking get request and remove it from channel, return current request data, if no request last data returned
	T b_get()	May-blocking get request, could be used in thread process only
sct_fifo_if	inherits sct_put_if<T> and sct_get_if<T>	
	void reset()	Call reset_put() and reset_get() both
	unsigned elem_num()	Number of elements in channel, value updated last clock edge for method, last DC for thread

Listing 1 contains Producer/Consumer illustrative example with Initiator and Target. In this example Producer contains method process which puts some data into Initiator *init*. Consumer has thread process which gets the data from Target *targ*. The Initiator and Target are bound to each other with function *bind()*.

```
struct Producer : public sc_module {
   sct_initiator<T> init{"init"};
   explicit Producer (const sc_module_name& name) : sc_module(name) {
      init.clk_nrst(clk, nrst);
      SC_METHOD(initProc); sensitive << init;
   }
   void initProc() {
      init.reset_put();
      if (init.ready()) init.put(getSomeValue());
   }};
struct Consumer : public sc_module {
   sct_target<T> targ{"targ"};
   explicit Consumer (const sc_module_name& name) : sc_module(name) {
      targ.clk_nrst(clk, nrst);
      SC_THREAD(targProc); sensitive << targ;
      async_reset_signal_is(nrst, false);
   }
   void targProc() {
      targ.reset_get();
      wait();
      while(true) {
         if (targ.request()) doSomething(targ.get());
         wait();
   }}};
struct Top : public sc_module {
   Producer prod{"prod"};
   Consumer cons{"cons"};
   explicit Top(const sc_module_name& name) : sc_module(name) {
      prod.init.bind(cons.targ);
   }};
```

Listing 1. Example of Initiator and Target usage

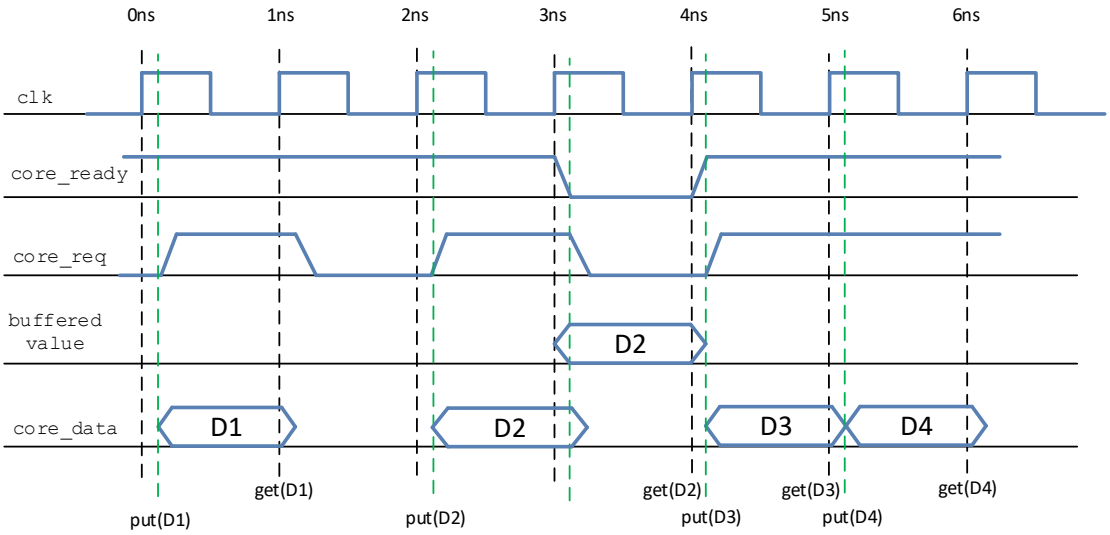

Figure 4. Initiator and Target signals

In Figure 4 there is a waveform for signals between Initiator and Target: *core_req*, *core_ready*, *core_data* and value stored in the Target buffer. This waveform matches to the previous example where put into Initiator is done in method process *initProc* and get from Target in thread process *targProc*. Gets from Target are done at the rising clock edge, puts to Initiator are done one delta cycle after that. At time 1ns *initProc* gets from Target, so *D1* is not buffered. At time 3ns *targProc* does not get from Target, therefore *D2* is stored in the internal buffer.

C. FIFO

FIFO can be used for inter-process communication between two processes in the same module and for storing requests inside one process. Also, FIFO could be used inside of Target as an extended buffer. FIFO implements *sct_fifo_if* interface. FIFO can have combinational or registered request and response paths which is specified with the constructor parameters. FIFO size is specified with the template parameter. To provide maximal throughput for various process types, FIFO should have enough size as it is described at [4].

Listing 2 contains FIFO illustrative example with thread process put to *fifo* and method process gets from *fifo*.

```
struct Top : public sc_module {
   sct_fifo<T, 10>      fifo{"fifo"};                 // FIFO size is 10 elements
   explicit Top(const sc_module_name& name) : sc_module(name) {
      fifo.clk_nrst(clk, nrst);
      SC_THREAD(producerProc); sensitive << fifo.PUT; // Process puts to FIFO
      async_reset_signal_is(nrst, 0);
      SC_METHOD(consumerProc); sensitive << fifo.GET; // Process gets from FIFO
   }

   void producerProc() {
      fifo.reset_put();
      wait();
      while (true) {
         fifo.b_put(getSomeValue());
         wait();
      }}

   void consumerProc() {
      fifo.reset_get();
      if (fifo.request()) doSomething(fifo.get());
   }};
```

Listing 2. Example of FIFO usage

D. Other Channels

Register channel is used to add state in SystemC method process. Register must be written in one method process and could be read in the same or other process. Register has *reset()*, *read()* and *write()* functions. Register can be read in one or multiple method or thread processes as well.

Listing 3 has Register illustrative example which accumulates data received from Target *targ*.

```
struct Top : public sc_module {
   sct_target<T>    targ{"targ"};
   sct_register<T>  cntr{"cntr"};
   explicit Top(const sc_module_name& name) : sc_module(name) {
      targ.clk_nrst(clk, nrst);
      cntr.clk_nrst(clk, nrst);
      SC_METHOD(cntrProc); sensitive << targ << cntr;
   }

   void cntrProc() {
      targ.reset_get();
      cntr.reset();
      if (targ.request()) cntr.write(cntr.read()+targ.get());
   }};
```

Listing 3. Example of Register usage

Pipeline register (*sct_pipe*) is intended to pipeline combinational logic and enable re-timing feature of a logic synthesis tool. Pipeline register can be used in one method or thread process as well as between two processes. Pipeline register is normally added to sensitivity list of a process where put or get is done. It supports to put bubbles and get backpressure. Pipeline register implements *sct_fifo_if*. Pipeline register size is specified with the template parameter.

Listing 4 has Pipeline register illustrative example with pipelining computation in method process.

```
struct Top : public sc_module {
   sct_target<T>     targ{"targ"};
   sct_initiator<T>  init{"init"};
   sct_pipe<T, 3>    pipe{"pipe"};            // Pipeline number of stages is 3
   explicit Top(const sc_module_name& name) : sc_module(name) {
      targ.clk_nrst(clk, nrst);
      init.clk_nrst(clk, nrst);
      pipe.clk_nrst(clk, nrst);
      SC_METHOD(pipeProc); sensitive << targ << init << pipe;
   }

   void pipeProc() {
      targ.reset_get();
      init.reset_put();
      pipe.reset();
      if (pipe.ready() && targ.request()) {
         T data = doSomething(targ.get());   // Heavy computation to be pipelined
         pipe.put(data);
      }
      if (pipe.request() && init.ready()) init.put(pipe.get());
   }
};
```

Listing 4. Example of Pipeline register usage

There are Signal (*sct_signal*) and Ports (*sct_in* and *sct_out*) in Single Source library which are similar to SystemC signal and ports. The reason to reimplement them in Single Source is to support Fast simulation mode.

Single Source clock (*sct_clock*) is implementation of clock source (generator) with generated events with an option to disable. In Cycle accurate mode *sct_clock* inherits *sc_clock* and has the same behavior. In Fast simulation mode *sct_clock* does not generate any events to speed up the simulation.

III. Evaluation Results

To evaluate the proposed methodology, we have taken several SystemC modules from industrial designs and rewritten them with Single Source library (we masked exact design names here). These modules include AXI port implementation, data multiplexor, address decoder, and on-die memory controller. In Table II there are comparison of line of code amount for original designs and designs with Single Source library and simulation speed up of Fast simulation mode versus Cycle accurate mode.

Table II. Evaluation results

Design	Original, LoC	Single Source library, LoC	Simulation speed up
A	360	191	7.9
B	517	287	6.0
C	1624	886	4.3
D	1411	1203	6.2
E	690	465	8.1
F	-	-	6.6

The proposed design methodology can be applied for arbitrary design kinds. The most simulation speed up can be achieved for designs with processes which have empty cycles, i.e., do not have input data to process at some cycles. In SystemC simulation, thread process is slower than method process, so more speed up effect can be achieved in designs dominated by thread processes.

To check correctness of the library channels implementation, we used about 300 unit tests and several real hardware designs with various complexity. We run the tests for SystemC implementation as well as for SystemVerilog generated with Intel Compiler for SystemC.

IV. Related Works

There are a few existing solutions for modelling and synthesis of hardware in SystemC language. The are proprietary libraries cynw_p2p, Flex Channels and others distributed with Stratus HLS tool [5]. We do not compare our library with them because of the license restrictions.

There are several open source SystemC component libraries. One of them is MatchLib library [6] which is available under Apache 2.0 license. SystemC designs with MatchLib channels can be synthesized with Catapult tool [7]. MatchLib includes four types of channels: Combinational, Bypass, Pipeline, and Buffer. These four channels differ in combinational request/response paths and storage capacity. All these four channel types are covered by Single Source Initiator/Target pair with options for combinational or buffered connection and for pipeline register in the request path.

MatchLib library provides input/output unbuffered and buffered ports to bind to the channels. The input ports have blocking and non-blocking pop functions, the output ports have the same push functions. In addition to that, buffered ports have *Full()* and *Empty()* functions. In Single Source library, ports are not required to bind modules at the same module hierarchy level (with the same parent module). Initiator/Target pair represents ports and channels both to reduce number of components and lines of the user code. To check if Initiator/Target connection is full/empty there are correspondent functions *ready()* and *request()*.

MatchLib connections designed with ports and channels can be used in clocked threads only. Important advantage of Single Source library is that the channels can be used in clocked thread and method processes as well.

Using Single Source channels in method processes allows to design zero latency modules. Generally, Intel Compiler for SystemC does not apply any rules or limitations on scheduling and translates SystemC code into equivalent SystemVerilog code as it is prescribed in [1]. That practically means, all the connections and channels options are valid for thread and method processes in any combinations.

Besides all that differences, the main advantage of the proposed solution is that the library and the high-level synthesis tool are open source.

There are two other SystemC libraries cited at systemc.org: SystemC-Components (SCC) [8] and Virtual Components Modeling Library (VCML) [9]. SCC is supposed to be a lightweight productivity library for SystemC and TLM 2.0 based modeling tasks. The Virtual Components Modeling Library contains a set of SystemC/TLM modeling primitives and component models that can be used to swiftly assemble system level simulators for embedded systems, i.e., virtual platforms.

SCC and VCML do not support hardware synthesis, while the proposed methodology has a feature that the model is synthesizable at every stage (architecture to RTL). The architecture model might be a highly abstracted version of the design similar to SCC and VCML. SCC and VCML use TLM debug and trace capture features. The proposed design methodology separates the synthesizable model and the features that provides debug and trace capture. Any external library or tool could be used for that purpose together with Single Source library.

V. CONCLUSION

In this paper we have presented the design methodology which uses a single source code in SystemC language for high-level modelling and hardware synthesis. The methodology is based on Single Source library which provides Fast simulation and Cycle accurate modes transparently for user code. Single Source library supports real-world design scenarios, uses simple design model, and protects architects and designers from common mistakes in inter-process communication. The proposed methodology reduces design time by ~2x with reducing the amount of user written code and provides ~4x-8x simulation speed up in Fast simulation mode.

REFERENCES

[1] “SystemC Synthesizable Subset Version 1.4.7”. http://accellera.org/downloads/standards/systemc.

[2] M. Moiseev, R. Popov, I. Klotchkov, “SystemC-to-Verilog Compiler: a productivity-focused tool for hardware design in cycle-accurate SystemC,” in proceedings of DvCon`19.

[3] “Intel Compiler for SystemC source code”. https://github.com/intel/systemc-compiler.

[4] “Single Source library wiki page”. https://github.com/intel/systemc-compiler/wiki/SingleSource-library.

[5] “Stratus High-Level Synthesis” https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[6] “MatchLib library source code”. https://github.com/NVlabs/matchlib.

[7] “Catapult C++/SystemC Synthesis tool” https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/.

[8] “SystemC-Components” https://github.com/Minres/SystemC-Components

[9] “Virtual Components Modeling Library” https://github.com/machineware-gmbh/vcml