SANDIA REPORT
SAND2021-12065
Unclassified Unlimited Release
Printed September 28, 2021

Sandia National Laboratories

ERAS: Enabling the Integration of Real-World Intellectual Properties (IPs) in Architectural Simulators

C. Hughes, A. Rodrigues, G. Voskuilen, K. S. Hemmert and S.D. Hammond
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM 87185
{chughes, afrodi, grvosku, kshemme, sdhammo}@sandia.gov

S. Nema, R. Razdan, D. Adak, and A. Awad
Secure and Advanced Computer Architecture Lab
NC State University
Raleigh, NC 27695
{snema, rrazdan2, dadak, ajawad}@ncsu.edu

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ERAS: Enabling the Integration of Real-World Intellectual Properties (IPs) in Architectural Simulators

S. Nema[1], R. Razdan[1], A. Rodrigues[2], K. S. Hemmert[2],
G. Voskuilen[2], D. Adak[1], S. Hammond[2], A. Awad[1], and C. Hughes[2]
[1]Secure and Advanced Computer Architecture Lab, NC State University, Raleigh, NC 27695
[2]Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185

SAND2021-12065

ABSTRACT

Sandia National Laboratories is investigating scalable architectural simulation capabilities with a focus on simulating and evaluating highly scalable supercomputers for high performance computing applications. There is a growing demand for RTL model integration to provide the capability to simulate customized node architectures and heterogeneous systems.

This report describes the first steps integrating the ESSENTial Signal Simulation Enabled by Netlist Transforms (ESSENT) tool with the Structural Simulation Toolkit (SST). ESSENT can emit C++ models from models written in FIRRTL to automatically generate components. The integration workflow will automatically generate the SST component and necessary interfaces to 'plug' the ESSENT model into the SST framework.

CONTENTS

LIST OF FIGURES

Nomenclature

HCL Hardware Construction Language

HDL Hardware Description Language

IP Intellectual Property

IR Intermediate Representation

RTL Register Transfer Language

SoC System-on-Chip

This page intentionally left blank.

1. INTRODUCTION

The ever-increasing complexity of computing systems, paired with new demands for customized and domain-specific architectures, motivate modular, reusable and open-source architectural components. For instance, today's system-on-chip (SoC) systems, which are used in nearly all aspects of life (e.g., automotive, edge devices, handheld devices, medical devices, cyber-physical systems etc.), are composed of tens to hundreds of modular components, potentially from different vendors. Such modularity has been a key enabler for agile development of SoCs for new products. Rather than spending thousands of person-hours developing new architectural components, modular design allows components to be licensed, bought, or reused while maintaining the ability to customize and optimize the architecture, domain-specific features, and software. The ability to customize hardware is driving the adoption of open-source instruction set architectures (e.g., RISC-V) by more and more companies as it enables them to develop custom processors that leverage a mix of internal and external Intellectual Properties (IPs) (e.g. memory controller, external cache controllers, etc.). Many applications from the Department of Energy (DOE) can potentially benefit from custom-designed processors, hence it is very important to be able to understand the performance impact of leveraging different IPs on the performance of workloads. Furthermore, understanding the impact of using future technologies with today's IPs is also enabled when there is an easy path to integrate the simulation models of third-party IPs in architectural models.

The Structural Simulation Toolkit (SST) [1] architectural simulator is being actively used to drive design space exploration of various architectural models and optimizations for hardware and software used (or will be used) by the DOE. The ability to augment the designs and architectural models explored in SST with low-level highly-accurate models, possibly with publicly available third-party IPs, can enable: (1) fast evaluation of the performance impact for alternative IPs in designs of interest; (2) enable system-level studies for using realistic IPs even before the technology or products are available; (3) save significant efforts for modeling architectural components, typically integrated as-is, by just leveraging low-level models already available; (4) facilitate more realistic design space exploration driven by real-world constraints of the used IPs (e.g., port width, interface protocol, interoperability, etc.); (5) accurate performance evaluations of sub-systems of interest and understanding possible limitations of alternative IPs/implementations rather than using abstracted medium-fidelity designs. Accordingly, our proposed framework, **ERAS**, enables seamless integration of low level designs into architectural simulators. Figure 1-1 depicts our vision for the future of architectural simulators and where ERAS framework fits in the architectural simulation workflow.

Enabling the integration of accurate cycle-level models generated from low-level designs is challenging. First, low-level designs and their corresponding models operate as real hardware insofar as they have ports and pins that must be precisely assigned to the correct binary values on each

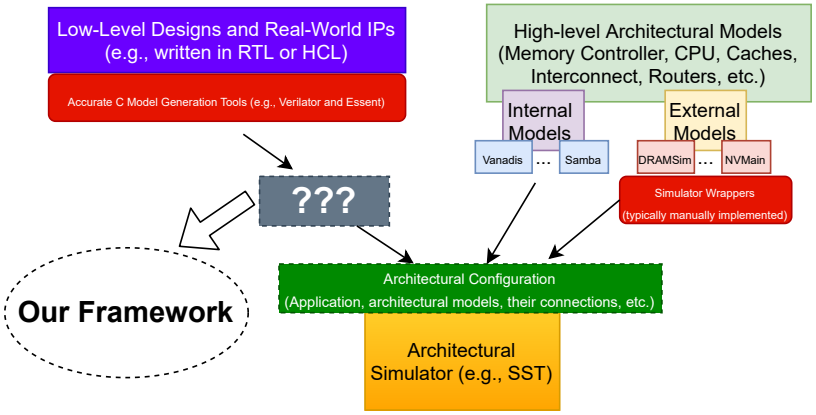

Figure 1-1. Software Architecture for Architectural Simulators Augmented with Real-World IPs

cycle. Meanwhile, most architectural simulators operate at the event granularity – completely ignoring the signaling requirements needed for correct operation of real designs. Second, different interfaces (e.g., AXI or PCI) require specific handling and commands; specific sets of signal values are encoded as different commands and hence the responses and signaling of these port must adhere to the protocol specifications. Third, each design might require a set of initial configuration parameters, which must be set at simulation start-up time. All of these challenges need to be addressed systematically to enable a versatile solution that can enable seamless integration of real designs models in architectural simulators.

To enable seamless integration of low-level designs and models written in Register Transfer Languages (RTL) or Hardware Construction Language (HCL) with SST's architectural components and models, we leveraged the ESSENT [2] toolchain to generate a C-model that is driven by stimuli and clocks (calling `eval` function). However, to allow seamless integration of such generated C-models, we built a framework and parser that produces a generic SST architectural component that is aware of the interfaces and ports required by the IP through a port-specification (PS) file. The PS file is a simple formatting file that defines the inputs of the IP and the interface type (e.g., using AXI or PCIe). By implementing the corresponding handlers for common interfaces, we can communicate messages (requests and responses) between the IPs' interfaces and the SST simulator. Moreover, we enable user-provided IP-specific parameters to be passed to each IP for appropriate configuration. The remainder of this document describes the proposed framework and provides a use case to demonstrate how to use the tools, followed by future work.

2. PROPOSED FRAMEWORK

In this section, we describe our framework and the changes required to enable the integration of low-level designs with architectural simulation models. The framework leverages the ESSENT tools to generate a C-model from a model described in FIRRTL [3], which is an Intermediate Representation (IR) for any Hardware Description Language (HDL). The hardware description, written in Chisel [4], which is a popular Hardware Construction language written is Scala [5], can be converted to FIRRTL using the Chisel compiler. Similarly, hardware descriptions written in Verilog [6] can be lowered to FIRRTL using Yosys [7]. The generated C-model is fed to the parser, the core of the framework, that provides the automation flow for interpreting user specifications and generating architectural components that can be integrated with with SST. Figure 2-1 shows the high-level systematic flow of the process of integration of low-level design with SST simulator.

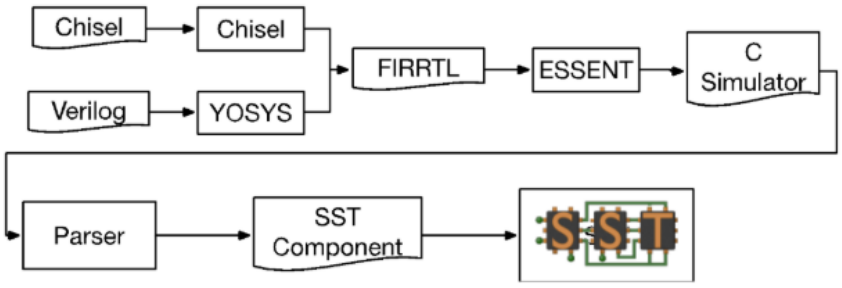

Figure 2-1. Systematic Flow – Integration of Low-level HDL Design with SST

The integration of RTL designs with an architectural simulator requires glue to bridge the gap between low-level cycle accurate details and the high-level models used by SST that often abstract away minute design details. This task is performed by the parser. The parser requires the following inputs in order to produce an SST component that can be loaded within SST as architectural unit or SST-Element:

1. External IP represented by a C-model generated using ESSENT
2. Generic SST component template where the details of the low-level design are filled to integrate with architectural simulator
3. Port-specification file that describes the inputs of the C-model to be updated on stimulus and a wrapper file that fills necessary boilerplate code and configurable parameters and statistics to be collected
4. Testbench, input stimulus, and test vectors to simulate the design

Section 3 describes the details of the user-specified requirements to the parser with examples and appropriate predefined format of the file in order for the parser to interpret content and insert code accordingly.

3. USER-SPECIFIED REQUIREMENTS TO ERAS

Figure 3-1 shows the input specifications that the parser requires to effectively generate a dynamically loadable SST component, shown in green. The C-model and generic component template, colored purple, are the fixed inputs that are provided to the parser directly without any modifications. User-provided inputs, yellow, are the port specification and wrapper files that are used to populate the generic template with the details of the generated C-model in order to produce design specific SST component.

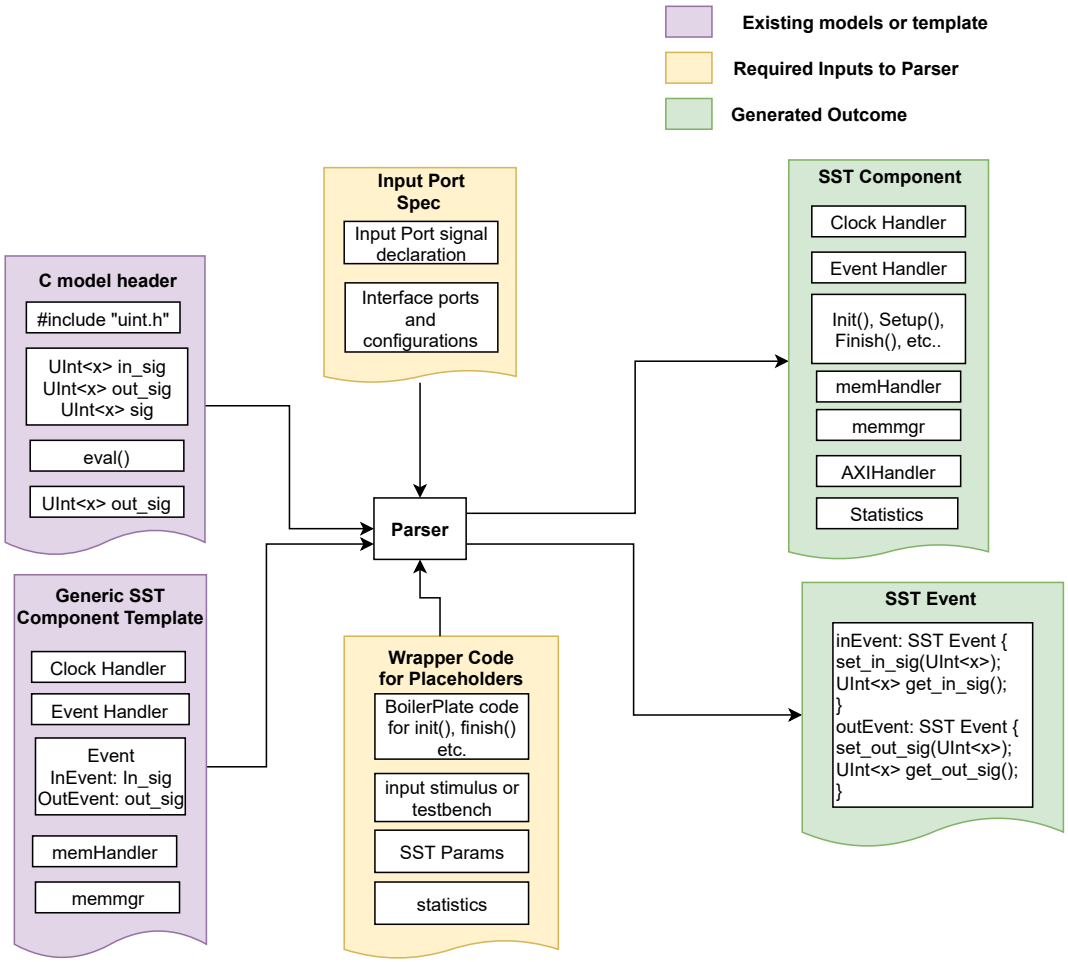

Figure 3-1. Block Diagram of ERAS Framework With Parser Input Requirements

3.1. C-model Header

The C-model is a header file produced from FIRRTL using ESSENT. This header must be included in the generic SST component template and is clocked every cycle by calling *eval()* API when integrated with SST.

3.2. Generic SST Component Template

This is the common template, structured as a SST component, that contains placeholders for various APIs that are updated based on design details specific to a particular C-model. It includes placeholders for various handlers, events and wrapper APIs along with hardened modules, such as the memory manager or memory handler. The details of specific module, categorized as either fixed or user-specific, are described as follows:

Fixed Modules

- *Clock Handler* calls *eval()* unless any extra logic is required
- *SST Links* and *ports* to Ariel and memH, Ariel Event, RTL Event to update input stimulus, few basic *SST Params*
- *memHandler* and associated APIs for generating and serving memory requests
- *memory manager* for address translations in case of memory requests by RTL model
- *AXI handler* for managing requests on AXI port of RTL model

User-Specified Modules

- *Clock Handler* for interface ports or any additional logic to be clocked
- *RTL Event* consisting input signals to be updated based on test vector or input stimulus
- *SST Parameters*, *Statistics*, additional ports (*SST_ELI_**) slot for configurable parameters from Python Config file
- *Boilerplate code* (*init()*, *setup()*, *finish()* APIs) for initialization or loading test vector/binary files for independent processors

3.3. Port-Specification (PS) and Wrapper Files

The input signal details and external ports of the RTL design are specified in the port specification file by the user *external_port.spec*. Further, the configuration of interfaces extended to the ports are also specified in it. For example, the AXI interface could be operating in streaming or burst mode and either in master or slave configuration. These details have to be specified to the parser to invoke the functionally appropriate handler to be interfaced with the RTL C-model. An example *external_ports.spec* is shown in Listing 3.1.

The wrapper file, *wrapper_code.spec*, contains the specification of statistics to be collected while simulating the RTL model, the external configuration knobs for tweaking design parameters, and other boilerplate APIs to initialize the model or logic to be executed while finishing the simulation.

```
#Full signal names along with complete alias path for external object reference (E.g. ldut.cbus.
    fixer.TLMonitor$$inst.plusarg_reader$$inst.out)
Input_sig:
io_ins_0
io_ins_1
io_ins_2
io_ins_3
io_load
io_shift

#Interface Ports floating out in RTL C-Model
AXIPort
#AXI Protocol to use. Will support AXIStream, AXI4, AXI4Lite
AXIProtocol:AXIStream
#Config - Master or Slave
config:Slave
#Mode - Streaming or Burst mode is supported. Current design supports only streaming mode
Mode:Streaming
#Interface Signals
AXI_sig:
io_nasti_aw_ready
io_nasti_ar_ready
io_nasti_w_ready
io_nasti_b_valid
io_nasti_b_bits_id
io_nasti_b_bits_resp
io_nasti_r_valid
io_nasti_r_bits_id
io_nasti_r_bits_resp
io_nasti_r_bits_last
io_nasti_r_bits_data
```

Listing 3.1 Example Port Specification File

Further, any extra logic to be added by parser in any Handler API or RTL event packet can be specified in the wrapper file. The wrapper is too large to embed here, but an examples are provided with the source code.

3.4. TestBench or Input Stimulus

Figure 3-2 demonstrates the ways to provide input stimulus to design. It can either be provided as a separate application binary that can leverage the existing CPU element in SST, Ariel, as the host device. Support for Ariel currently exists within the generic component that instruments the binary to provide stimulus to the RTL C-model. The stimulus can also be directly provided as simple assignment statements in *wrapper_code.spec*:

1. Event Handler, *RTLEvent()*, API in case of external stimuli triggered when RTL Event from CPU or other element arrives
2. Initialization, *init()*, API function in case a hex file has to be loaded to memory and executed instruction-by-instruction at each cycle as in case of RISC-V CPUs

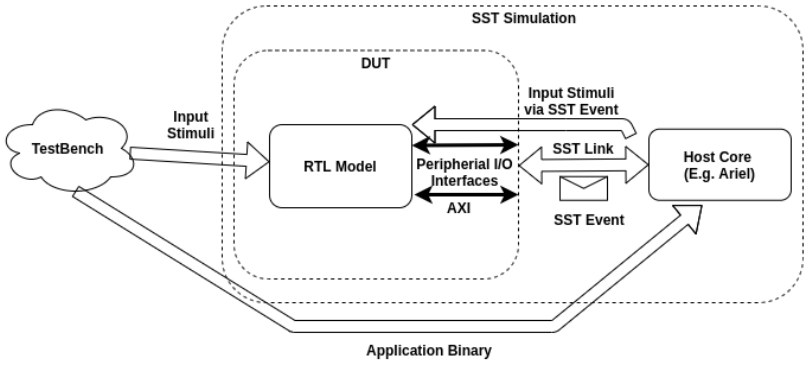

Figure 3-2. Ways to provide test vector or input stimuli to simulate RTL model

To provide it via application binary, the test harness or workload application capable of running onto the design should be compiled to x86 and the path should be specified in the Python configuration file. The next section provides an example use-case by integrating a simple design described in Chisel integrated and simulated over SST.

4. EXAMPLE USECASE

We show an example use case integrating a low-level design described in Chisel and converting it to a C-model by following the process described in Section 2 and integrating with SST in a step-wise manner. Also, in this section, we delve into the process necessary to compile a design written in Chisel, converting it to a C-model, and then manually integrating with SST, fulfilling the user requirements specified in section 3 since the automated parser flow has not been completed yet.

We chose a toy design, *Vector Shift Register*, written in Chisel as the starting point. The C-model is generated from the Chisel HDL following the steps below in Sections 4.1.1 and 4.1.2.

4.1. Generating C-model

The next two sections discuss how to covert a low-level implementation to a high-level C-model.

4.1.1. Chisel HDL to FIRRTL

It is essential to have Scala project build files *(build.sbt, build.properties)* for large projects in the source folder to compile the design and generate the correct FIRRTL. Though most designs have their own project build file for large projects, the steps to build a simple Chisel project and generate FIRRTL are:

Compilation step for sbt: Install the following prerequisites:

- https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html
- JVM(jdk) compatible with scala 2.12.4 (JAVA 11 JDK), javac
- scala
- scalac

Add these lines at the end of Scala sources:

```
object YourModuleDriver extends App {
  Chisel3.Driver.execute(args, () => new YourModule)}
```

Replacing *"YourModule"* by the name of your module. And add a build.sbt file in the same directory of your sources with these lines:

```
scalaVersion := "2.11.8"
resolvers ++= Seq( Resolver.sonatypeRepo("snapshots"), Resolver.
    sonatypeRepo("releases") )
libraryDependencies += "edu.berkeley.cs" %% "Chisel3" % "3.0-
    SNAPSHOT"
```

To generate FIRRTL and Verilog:

```
$ sbt "run-main YourModuleDriver"
```

The FIRRTL (yourmodule.fir) will be created in the source directory. Additional Chisel designs are available here: https://github.com/ucb-bar/Chisel-tutorial

For the shift register example discussed above, the FIRRTL can be generated using this command:

```
$ sbt > test:runMain solutions.Launcher VecShiftRegister
```

The generated FIRRTL file will be located in *test_run_dir*, which is generally the directory where ChiselTest (test harness for Chisel-based RTL designs) based tests are located.

4.1.2. FIRRTL to C-model Header

FIRRTL to C-model header conversion the requires the *ESSENT* C-simulator tool [2]. ESSENT (ESSENTial signal simulation enabled by netlist transformations) is a high-performance RTL simulator generator. Given a hardware design in FIRRTL, ESSENT emits C++ that can be compiled to make a fast simulator of the design. ESSENT provides several optimizations to improve performance, and they can be turned on or off with command-line flags. Use ESSENT to generate a single header file (.h) that represents the C-model of the RTL design by doing the following:

```
git clone https://github.com/ucsc-vama/ESSENT.git
cd ESSENT
sbt assembly
cd utils/bin/
./ESSENT -O1 <path_to_generated_fir_file>/my_design.fir
```

4.2. Integration of C-model with SST

Many of the steps to integrate C-model with SST, which are described below, will eventually be automated by the parser. However, the steps mentioned in the user requirements section have to be provided by the user that the parser can interpret for the code generation process of final SST Component. Integration of the C-model with SST mainly requires code insertion in the Generic SST component template which consist of placeholders for various APIs that are updated by parser in automated fashion based on user specifications. Since the current integration doesn't support automation, the placeholders have been manually updated in the VecShiftRegister example.[1]

Generic Component Path: `sst-elements/src/sst/elements/rtlcomponent/rtlcomponent.h`
Example VecShiftReg Path: `sst-elements/src/sst/elements/sst-external-element/vecshiftreg.h`

[1] Repository access: http://github.com/shubhamn94/sst-elements

4.2.1. Including ESSENT Header in the Generic Component

The ESSENT generated header described in the previous section has to be included in the generic SST Component template as shown in Figure 4-1.

```
#include <sst/core/component.h>
#include <sst/core/interfaces/simpleMem.h>
#include <sst/core/timeConverter.h>

//Header file will be changed to the RTL C-model under test
#include "rtl_header.h"
#include <sst/core/link.h>
#include <sst/core/clock.h>
#include <sst/core/eli/elementinfo.h>
#include "rtlevent.h"
#include "arielrtlev.h"
#include "rtlreadev.h"
#include "rtlwriteev.h"
#include "rtlmemmgr.h"
#include "AXI_port.h"
```

Figure 4-1. Generated Header Included in SST Component

- `rtl_header.h` is the header for RTL model. The header name will be replaced by the user-specific C-model header name, e.g. VecShiftRegister.h
- `AXI_port.h` is the C-model header for an AXI controller fixed for an AXI handler in SST

The logic in the C-model exercised by the *eval()* API executes at every cycle, controlled by the *ClockTick()* API called by Clock Handler at every cycle. Any extra wrapper logic around *eval()* must be manually added until the automation flow is complete.

4.2.2. Updating RTL Port Signals in RTLEvent

The C-model generated from a Chisel or Verilog description consists of various peripheral ports to interact with external components communicating via peripheral interface protocols such as low-speed protocols (e.g. UART, SPI, AXI) and high-speed interfaces protocols (e.g. PCIe). However, the architectural models in the SST framework communicate through abstract interfaces (SST Link) and abstract messaging packets (SST Events) that trigger a specific behavior in a component. To bridge this gap between low-level signaling schemes and na abstract module-level interaction, the messages from SST Events have to be decoded and converted into valid signaling schemes that are applied to the input ports of the low-level RTL model. The applied external stimulus at the input ports of RTL model are consumed by the model to produce specific functional behavior during simulation. This task is carried out by special APIs referred to as *Interface Handlers* within the SST Component. Similarly, any change in the external interface port of the RTL model is sensed by the polling logic present in the clock handler, clocked at every cycle, invoking the corresponding Interface Handler to interpret the message signal and converting it into a SST Event for communication with other SST components.

The RTL port signals are updated in the `RTLEvent` class, called by the `RTLEvent` Handler. The testbench or test vector can call the `RTLEvent` Handler to provide periodic stimulus to design

```
52
53  void RTLEvent::input_sigs(Rtlheader* cmodel) {
54
55      cmodel->reset = UInt<1>(1);
56      UInt<4>* rtl_inp_ptr = (UInt<4>*)inp_ptr;
57      cmodel->io_ins_0 = rtl_inp_ptr[0];
58      cmodel->io_ins_1 = rtl_inp_ptr[1];
59      cmodel->io_ins_2 = rtl_inp_ptr[2];
60      cmodel->io_ins_3 = rtl_inp_ptr[3];
61      cmodel->reset = UInt<1>(0);
62      fprintf(stderr, "\ninput_sigs: %" PRIu8, cmodel->io_ins_0);
63      fprintf(stderr, "\ninput_sigs: %" PRIu8, cmodel->io_ins_1);
64      fprintf(stderr, "\ninput_sigs: %" PRIu8, cmodel->io_ins_2);
65      fprintf(stderr, "\ninput_sigs: %" PRIu8, cmodel->io_ins_3);
66      return;
67  }
68
69  void RTLEvent::control_sigs(Rtlheader* cmodel) {
70
71      fprintf(stderr, "\nctrl_sigs called");
72      cmodel->reset = UInt<1>(1);
73      UInt<1>* rtl_ctrl_ptr = (UInt<1>*)ctrl_ptr;
74      cmodel->io_shift = rtl_ctrl_ptr[0];
75      cmodel->io_load = rtl_ctrl_ptr[1];
76      cmodel->reset = UInt<1>(0);
77      fprintf(stderr, "\nctrl_sigs %" PRIu8, cmodel->io_shift);
78      fprintf(stderr, "\nctrl_sigs %" PRIu8, cmodel->io_load);
79      return;
80  }
81
```

Figure 4-2. RTL Model Signal Updates Based on Stimulus

while simulation. Until the automation flow is complete, manual update in the placeholders for input and control signals of C-model is required. Figure 4-2 highlights the portion added for the *VecShiftRegister* example and the same should be replaced by corresponding input port signals of any other C-model supposed to be integrated with SST. These modifications will eventually happen through the automated flow parser by parsing `external_port.spec` and `wrapper_code.spec` specification files, provided by the user.

4.2.3. SST Component Wrapper Code for RTL C-model

Wrapper or Boilerplate code holds the configuration and linking logic necessary for wrapping the C-model effectively within SST. This includes specifying initialization, setup and completion steps for C-model, if any, before it's execution begins. Further, the statistics to be collected, configuration parameters and additional SST ports and links to be established with other SST elements apart from memHierarchy and CPU can be specified in the wrapper code. Also, any extra logic in the Event handler or Clock Handler necessary for correct functional execution can be inserted in the corresponding provided APIs in Generic SST Component. As of now, all modifications steps are manual. However, in the automation flow these API updates will be carried out by parsing the user specifications provided in `wrapper_code.spec` file. Sample `wrapper_code.spec` file with few placeholders filled is provided here: https://github.com/shubhamn94/sst-elements/blob/shubham/src/sst/elements/rtlcomponent/wrapper_code.spec. The APIs to be updated are mentioned in section 3.2.

4.2.4. Input Stimulus to RTL Model

To observe the functional behavior of the RTL model, a test harness must be provided. It can be either provided via the application binary or by updating the wrapper APIs and calling it at specific intervals as per the test harness. In the *VecShiftRegister* example, the application binary is provided to the host, which is Ariel in this example. Ariel relies on Intel's PIN tool to dynamically instrument the user-specific APIs, *start_RTL_sim()* and *Update_RTL_signals()*, to generate RTL Events that provide the stimulus.

```
65    io_shift = UInt<1>(1);
66    io_ins_0 = UInt<4>(9);
67    io_ins_1 = UInt<4>(4);
68    io_ins_2 = UInt<4>(2);
69    io_ins_3 = UInt<4>(7);
70    io_load = UInt<1>(0);
71
72    size_t inp_size = sizeof(UInt<4>) * 5;
73    size_t ctrl_size = sizeof(UInt<1>) * 2;
74    RTL_shmem_info *shmem = new RTL_shmem_info(inp_size, ctrl_size);
75
76    inp_ptr = (UInt<4>*)shmem->get_inp_ptr();
77    ctrl_ptr = (UInt<1>*)shmem->get_ctrl_ptr();
78
79    ctrl_ptr[0] = io_shift;
80    ctrl_ptr[1] = io_load;
81    inp_ptr[0] = io_ins_0;
82    inp_ptr[1] = io_ins_1;
83    inp_ptr[2] = io_ins_2;
84    inp_ptr[3] = io_ins_3;
85
86    Update_RTL_Params *params = new Update_RTL_Params();
87    params->storetomem(shmem);
88    params->check(shmem);
89    ctrl_ptr[0] = io_shift;
90    ctrl_ptr[1] = io_load;
91    inp_ptr[0] = io_ins_0;
92    inp_ptr[1] = io_ins_1;
93    inp_ptr[2] = io_ins_2;
94    inp_ptr[3] = io_ins_3;
95    params->storetomem(shmem);
96
97    //fprintf(stderr, "\nGoing to call start RTL sim");
98    start_RTL_sim(shmem);
99    bool *check = (bool *)shmem->get_inp_ptr();
100   //fprintf(stderr, "\n\n Checking mlmstream: %d\n", *check);
```

Figure 4-3. Application to Provide Input Stimulus to RTL Model

The workload or application with user-specific APIs to initiate events is compiled and the executable is loaded by the Python configuration file. Figure 4-3 shows the source code for an example application with user APIs to generate stimulus for the *VecShiftRegister* example.

4.2.5. Compilation and Configuration Changes in SST

This section describes how to compile the new SST component. In the future, the compilation steps detailed in this section will be part of the automated flow of the ERAS framework and will not have to be performed manually.

The steps to compile the component and update the Python configuration file are as follows. In this example, the name of the files `rtlcomponent` and `rtlevent` are changed. These must also be updated in AutoMake (`sst-elements/src/sst/elements/rtlcomponent/Makefile.am`) in the `rtlcomponent` directory. The name of the folder should match the name of the file, `rtlcomponent.cc`.

Replace `"rtlcomponent"` in all the places in the AutoMake file `Makefile.am` with the updated name of the file `rtlcomponent.cc` then build SST Elements:

```
cd sst-elements/
make && make install
```

This will build and register the modified generic SST component to a dynamically loadable component in SST via the config file. The `sst-info` command will show whether the created generated final RTL component is registered with SST.

```
sst-info
```

After performing the compilation steps described in section 4.2.5, the dynamically loadable component of the RTL model is generated, which can be swiftly integrated with the rest of the SST system.

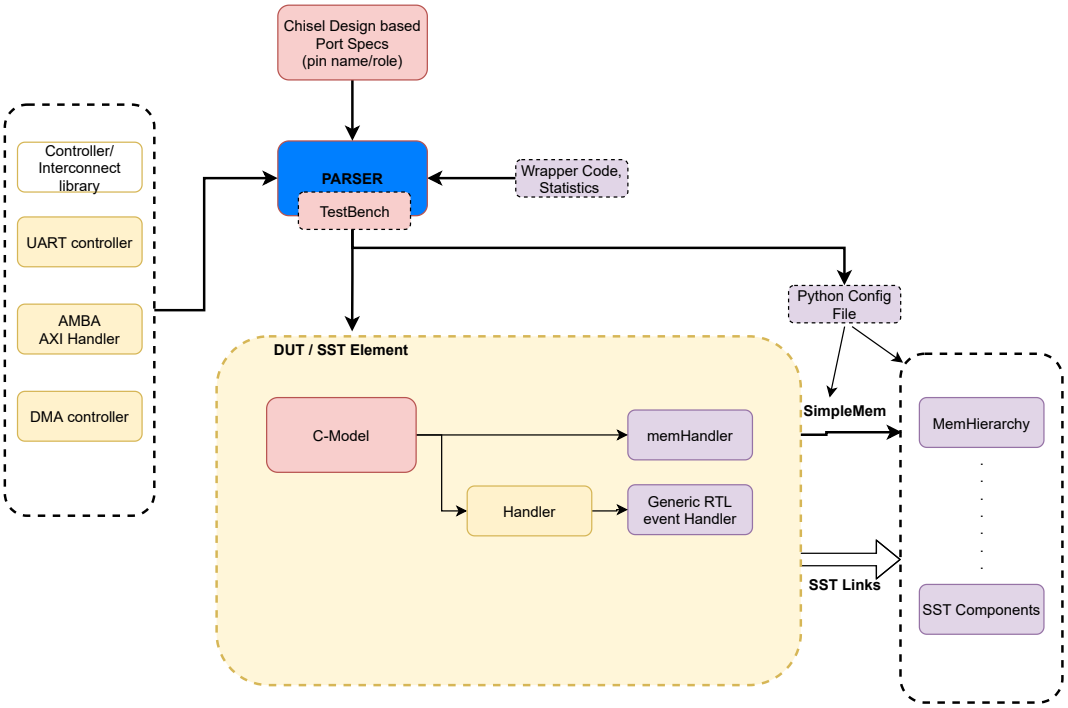

Figure 4-4. Functional Block Diagram of ERAS Integrated With SST

Figure 4-4 shows the functional block diagram of the framework and the detailed architectural modules in the generated SST Component. The final SST component, or Design Under Test (DUT), consists of an ESSENT-generated C-model interfaced with various Handlers – specifically memHandler and an interface handler (e.g. AXI Controller represented by yellow block) – to communicate with the rest of the elements in the SST system using SST Events. The handler to be invoked using the handler library shown in the leftmost block in Figure 4-4 will depend on the

user specifications provided to Parser in *PS* file. The generated component by Parser can be loaded in Python configuration file and can be connected with rest of the SST Elements using SST Links in a similar fashion as any other element and statistics to be collected can also be specified in it.

The Python configuration must be updated to provide the path to executable, as shown in the Figure 4-5. Instantiate the SST Component of type *vecshiftregister* and add the necessary configuration parameters. The memory manager must also be included as a sub-component and can be invoked after instantiating the RTL component.

```
app = sst_workdir+"/sst-tools/tools/ariel/femlm/examples/stream/mlmstream"

if not os.path.exists(app):
    app = os.getenv( "OMP_EXE" )
    printf("OS PATH DOESN'T EXIST")
ariel = sst.Component("A0", "ariel.ariel")
ariel.addParams({
        "verbose" : "1",
        "maxcorequeue" : "256",
        "maxissuepercycle" : "2",
        "pipetimeout" : "0",
        "executable" : app,
        "arielmode" : "1",
        "clock" : "1GHz",
        "arielinterceptcalls" : "1",
        "launchparamcount" : 1,
        "writepayloadtrace" : 1,
        "launchparam0" : "-ifeellucky",
        })

memmgr = ariel.setSubComponent("memmgr", "ariel.MemoryManagerSimple")

corecount = 1;

rtl = sst.Component("vecshiftregister", "vecshiftreg.vecShiftReg")
rtl.addParams({
        "ExecFreq" : "1GHz",
        "maxCycles" : "100"
        })

rtlmemmgr = rtl.setSubComponent("memmgr", "vecshiftregister.MemoryManagerSimple")

rtl_cache_link = sst.Link("rtl_cache_link")
rtl_cache_link.connect( (rtl, "RtlCacheLink", "50ps"), (l1rtlcache, "high_network_0", "50ps") )

cpu_rtl_link = sst.Link("cpu_rtl_link")
cpu_rtl_link.connect( (ariel, "rtl_link_0", "50ps"), (rtl, "ArielRtllink", "50ps") )
```

Figure 4-5. Python Configuration Example

4.2.6. SST Simulation

The simulation can be initiated by calling the Python file through SST as shown below for the *VecShiftRegister* example. Selected statistics in the Python configuration file will be collected and presented at the end of simulation.

```
cd sst-elements/src/sst/elements/rtlcomponent/stream
sst runstream.py
```

5. FUTURE WORK

This section describes the subsequent efforts to be carried out to build the automation flow for the ERAS framework. This includes some necessary experiments to be carried out with complex examples, such as integrating a RISC-V based System on Chip (e.g RocketChip [8]) with a full system and performing simulation in order to verify end-to-end full system simulation flow. This will rule-out any design considerations or requirements that may have been missed or overlooked while developing the integration flow. The subsequent steps being taken to effectively integrate complex designs with SST are:

- Disintegrating Complex RTL model into various functional submodules and coupling those with the interface handlers supported in the ERAS framework to perform full system simulation while collecting inter-module transaction statistics. Current efforts are being made to decouple AXI interface ports within the RocketChip Chisel description and interface it with AXI handler C-model in a slave configuration, which is integrated within SST.
- Initializing RTL models, specifically processor cores, and loading the executable to memHierarchy within SST while dynamically invoking the SST Component instead of relying on external host core, such as Ariel, to feed stimulus to design under test.
- Expanding the interface library of the ERAS framework to exhaustively support various combination of AXI modes and configurations along with handler support for other interfaces (e.g. UART) to expand the range of external IPs that can be effectively integrated with SST.

Summarizing the automation efforts that will eliminate most of the manual updates to generic component being carried out currently:

1. The *PS* will be consumed by the parser and parsed to extract logic to be inserted into various portions of Generic SST component. This includes:
   a) Input port signals specified will be inserted and updated in RTLEvent whenever an stimulus triggers event handler.
   b) Interface handlers will be invoked based on the configurations specified in external_port.spec file and will be coupled with C-model in clockTick() API of generic RTL component.
   c) All interface handlers, *e.g AXI*, will have two front. RTL - AXI:raw_data - SST and SST:raw_data:AXI - RTL for two-way communication with RTL model. The AXI specific signals in each handler will vary depending on the AXI configurations and input signal references/naming convention specified in external_port.spec file. The logic to couple interface handler with RTL model will be added in clockTick() API along with eval() API of interface C-model controller.

2. The wrapper_code.spec will be consumed by the parser and parsed to extract logic to be inserted into various portions of Generic SST component. This includes:

    a) Statistics to be recorded and SST configuration parameters to be tweaked will be specified in Python configuration file. Based on the trigger events specified corresponding to the statistics in wrapper_code.spec stat variables will be sprinkled in the generated SST component and incremented.

    b) BoilerPlate code for APIs *(e.g. setup(), init(), finish())* will be extracted by parser and inserted into generated RTL Component. Further, any extra logic to be inserted in hardened APIs such as *ClockTick()*, *handleRtlEvent()* will be added by parser based on the directions in *wrapper_code.spec*

3. The manual compilation steps to generate the C-model from ESSENT and compile the SST component will be automated through simple scripts. Also, compilation for an application binary or a testbench to be executed could be included automation. Later, efforts can also be made to automatically generate Python config file with the configuration specified in plain text in specification file.

6. CONCLUSION

This report introduced a new framework, ERAS, for integrating low-level RTL/HDL models with the architectural models available in the Structural Simulation Toolkit. This framework will allow researchers to perform node- and system-level design studies using mixed-fidelity models and to explore the emerging open-source hardware landscape. The example provided in this report is just a starting point, further research and engineering efforts will enable the integration of more complex models and automated design integration.

REFERENCES

[1] A. Rodrigues, R. Murphy, P. Kogge, and K. Underwood, "The Structural Simulation Toolkit: A Tool for Bridging the Architectural/Microarchitectural Evaluation Gap," 2004.

[2] S. Beamer and D. Donofrio, "Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20, IEEE Press, 2020.

[3] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware Construction Languages, Compiler Frameworks, and Transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 209–216, Nov 2017.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

[5] P. M. Odersky, "The Scala Programming Language."

[6] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006.

[7] C. Wolf, "Design and Implementation of the Yosys Open Synthesis Suite," tech. rep., 2013.

[8] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

DISTRIBUTION

Email—External [redacted]

Name	Company Email Address	Company Name
Amro Awad	ajawad@ncsu.edum	NC State University

Email—Internal [redacted]

Name	Org.	Sandia Email Address
Simon D. Hammond	01422	sdhammo@sandia.gov
Clay Hughes	01422	chughes@sandia.gov
Technical Library	1911	sanddocs@sandia.gov

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.