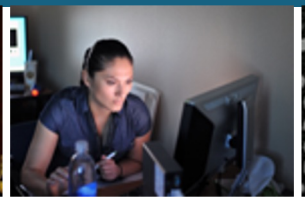
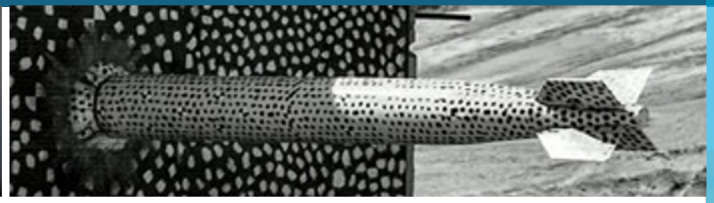
ERAS: Enabling Integration of Real-World Intellectual Properties in Architectural Simulators


PRESENTED BY

The SST and SACA Teams (Sandia, NC State)

ISPASS 2023 TUTORIAL

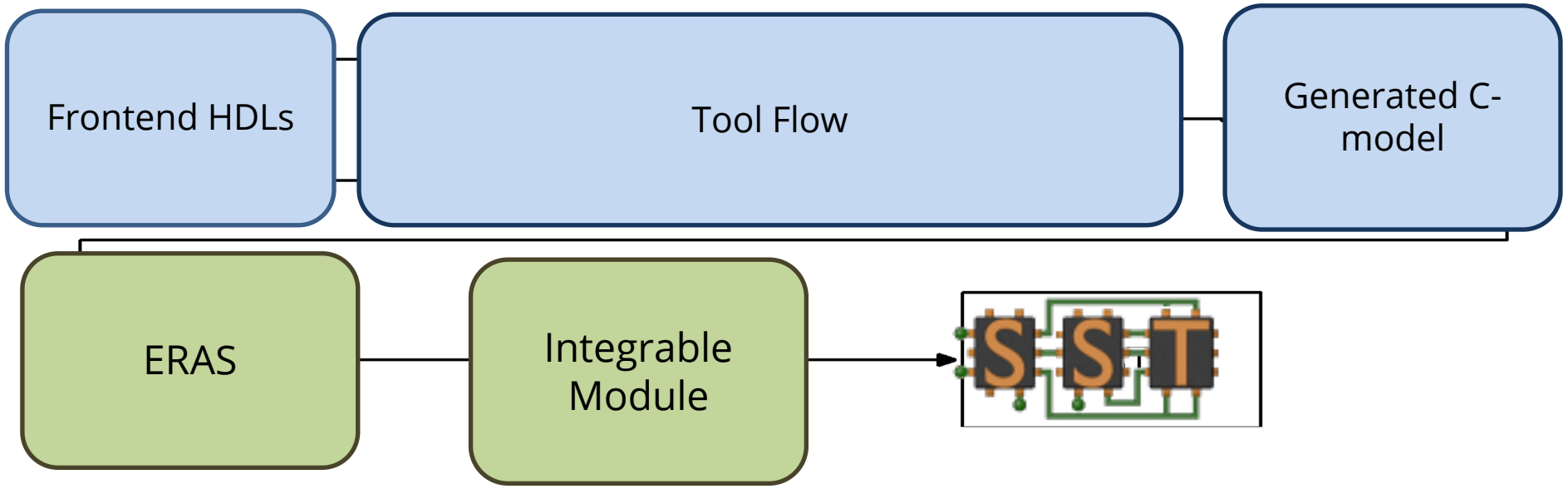
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2023-02523C

Outline

- **Overview**
- **Motivation**
- **Framework Highlight**
- **Parser**
- **Handlers**
- **Summary**
- **Future Work**

Overview



- General WorkFlow: HDL → FIRRTL → ESSENT → **Parser** → SST
- **Goals:**
  - Support several types of Low-level IPs
  - Easy, customizable workflow

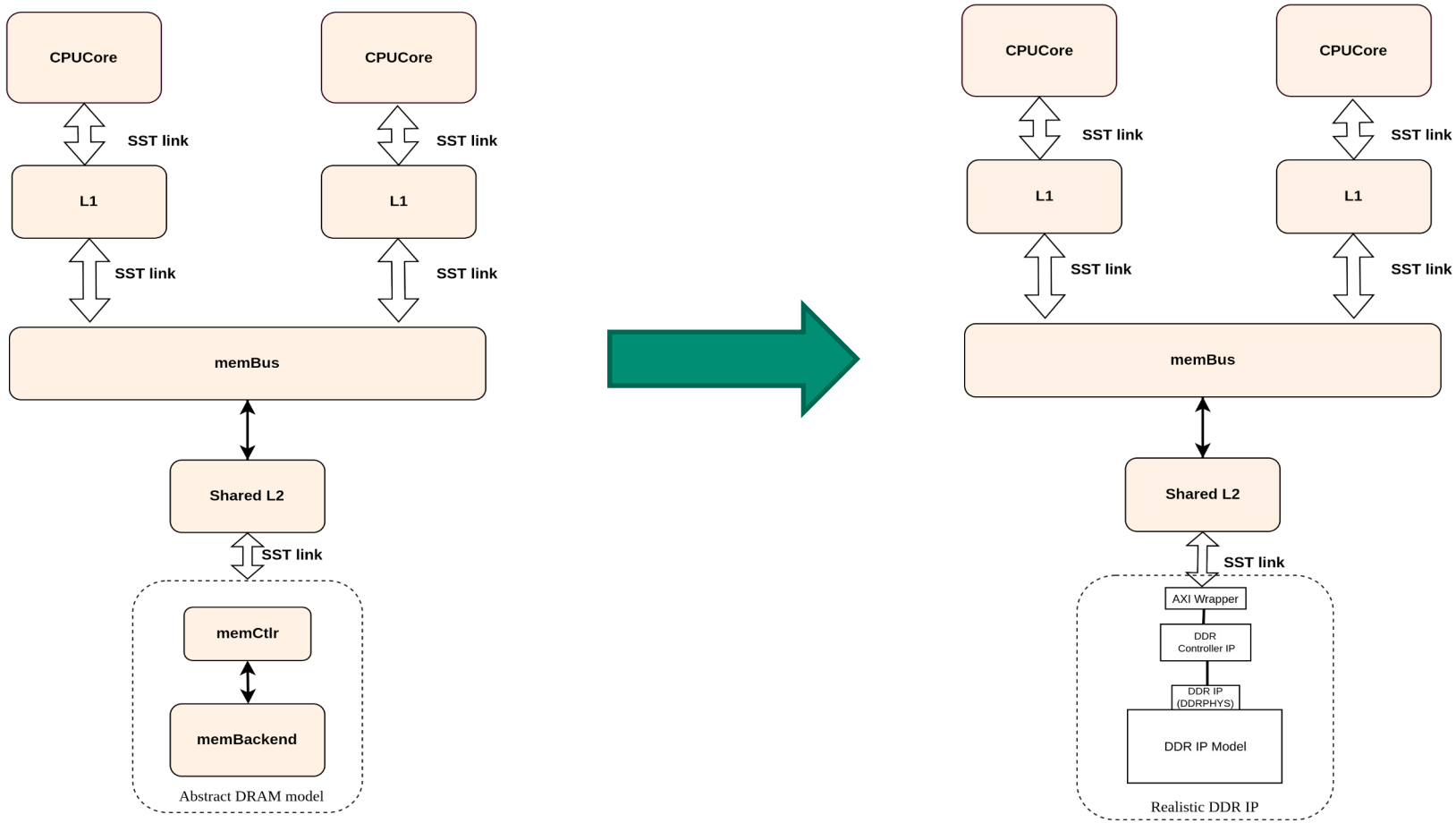
Outline

- Overview
- **Motivation**
- Framework Highlight
- Parser
- Handlers
- Summary
- Future Work

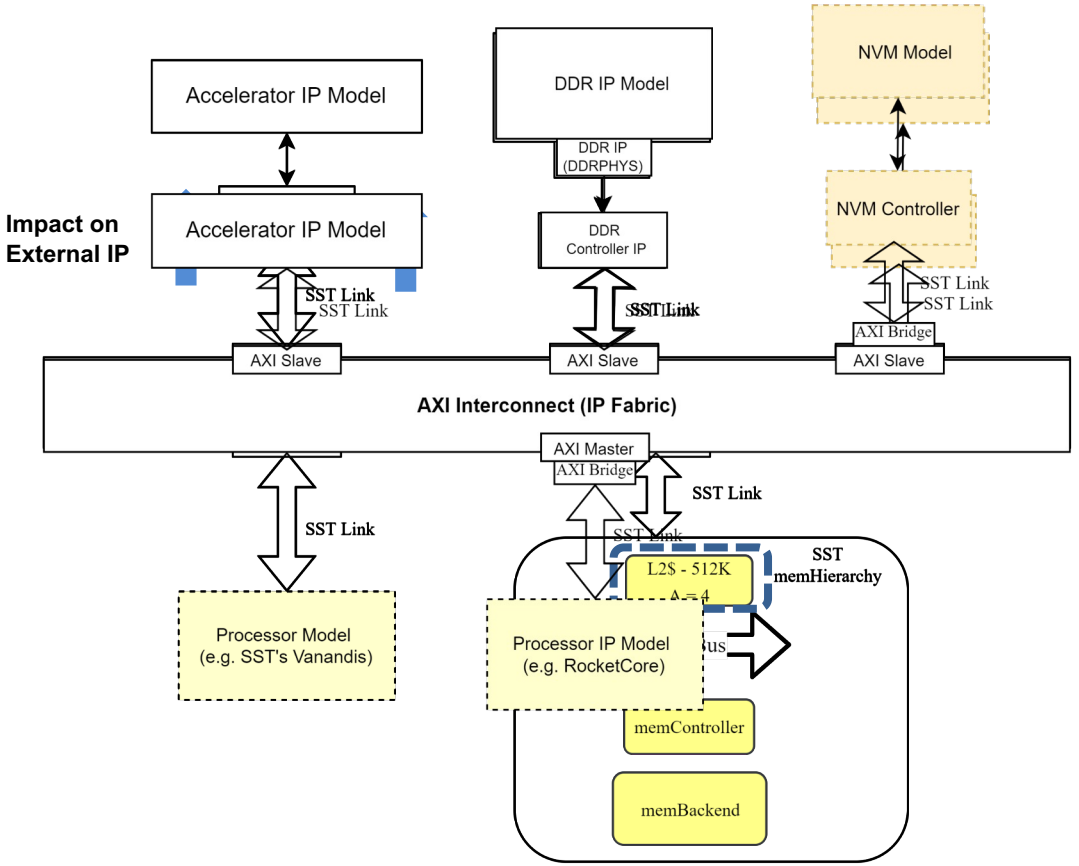
Why ERAS ?

- Analyze impact on system level behavior with realistic cycle-accurate low level IP model
- Analyze impact of system level changes on external IP integrated with SST
- Extensibility to an extent that full system is modelled through only realistic IPs

Why ERAS ?

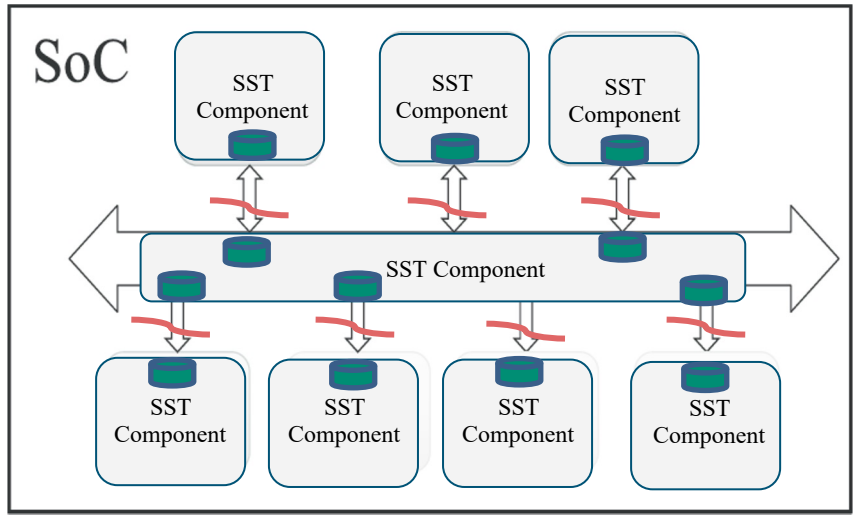

CPUCore
CPUCore
SST link
SST link
L1
L1
SST link
SST link
memBus
Shared L2
SST link
memCtlr
memBackend
Abstract DRAM model
CPUCore
CPUCore
SST link
SST link
L1
L1
SST link
SST link
memBus
Shared L2
SST link
AXI Wrapper
DDR Controller IP
DDR IP (DDRPHYS)
DDR IP Model
Realistic DDR IP

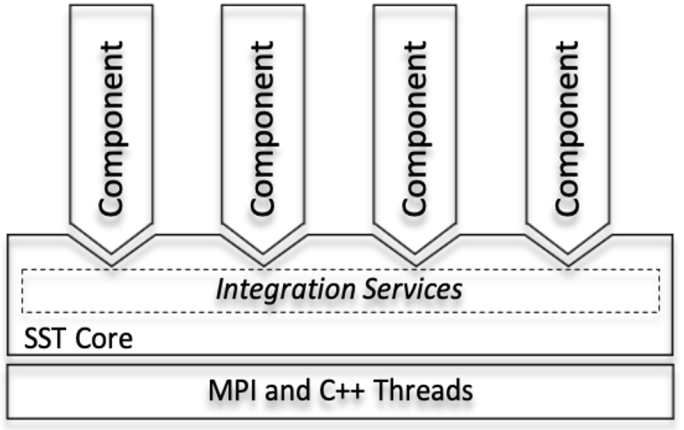
Why ERAS ?


Impact on
External IP
Accelerator IP Model
Accelerator IP Model
SST Link
SST Link
AXI Slave
DDR IP Model
DDR IP
(DDRPHYS)
DDR
Controller IP
SST Link
AXI Slave
NVM Model
NVM Controller
SST Link
SST Link
AXI Bridge
AXI Slave
AXI Interconnect (IP Fabric)
AXI Master
AXI Bridge
SST Link
SST Link
Processor Model
(e.g. SST's Vanandis)
SST
memHierarchy
L2$ - 512K
Processor IP Model
(e.g. RocketCore)
memController
memBackend

Why ERAS ?

- Analyze impact on system level behavior with realistic cycle-accurate low level IP model
- Analyze impact of system level changes on external IP integrated with SST
- Extensibility to an extent that full system is modelled through only realistic IPs



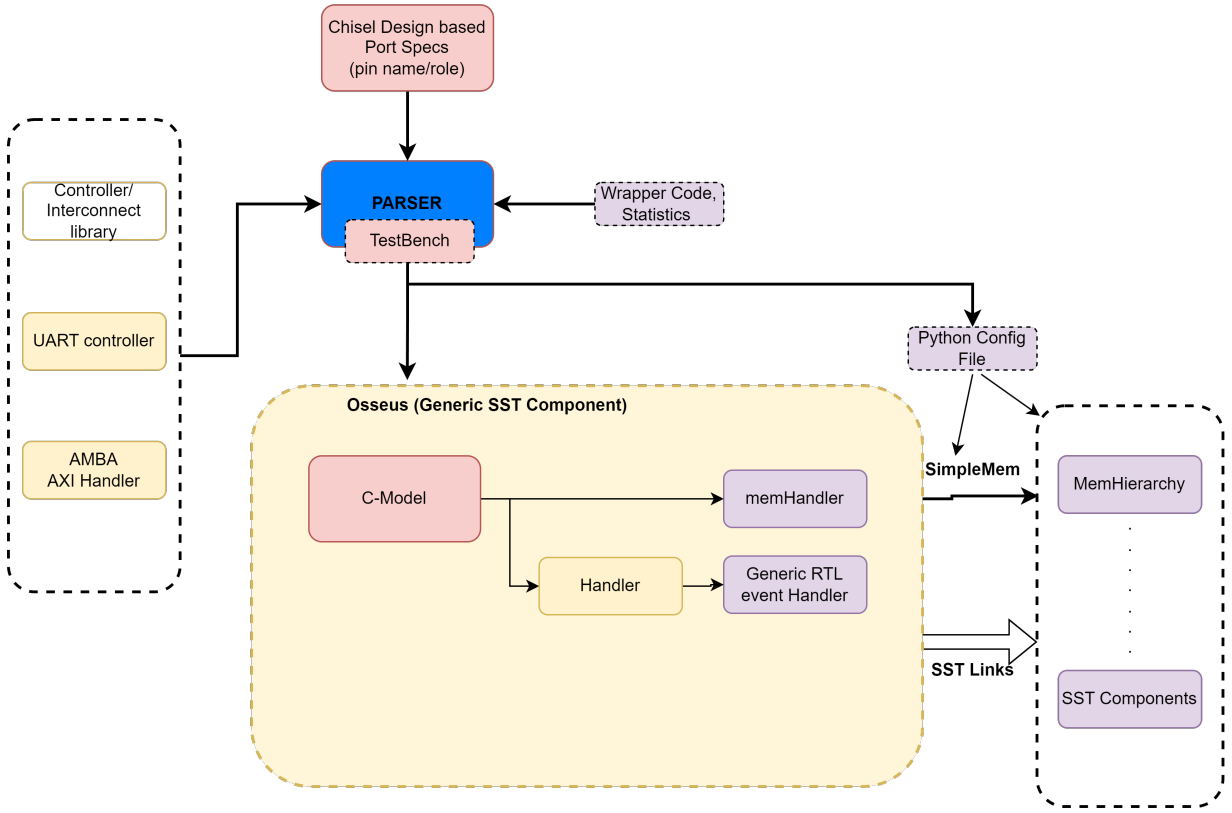


Outline

- Overview
- Motivation
- **Framework Highlight**
- Parser
- Handlers
- Summary
- Future Work

Framework Highlight

- Designer provides necessary information to the Parser
- **Parser** interprets the information and invoke necessary blocks of the generic component, referred as Osseus, to integrate the C-model with SST

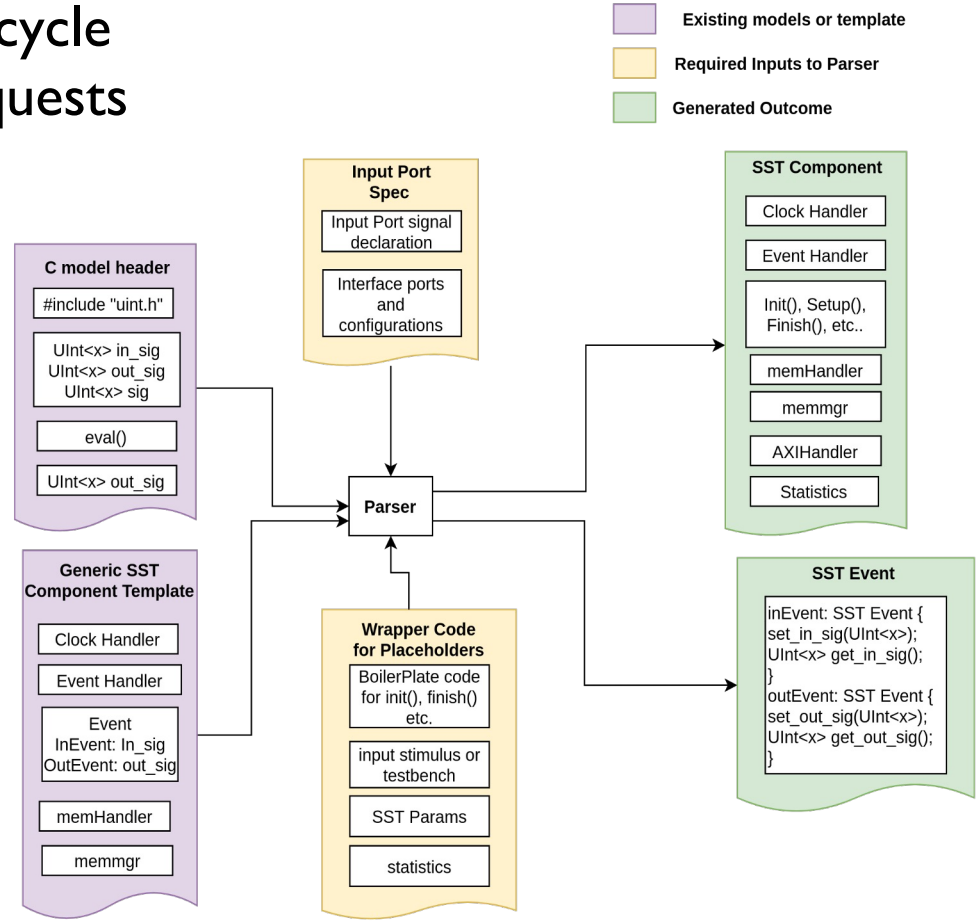


Outline

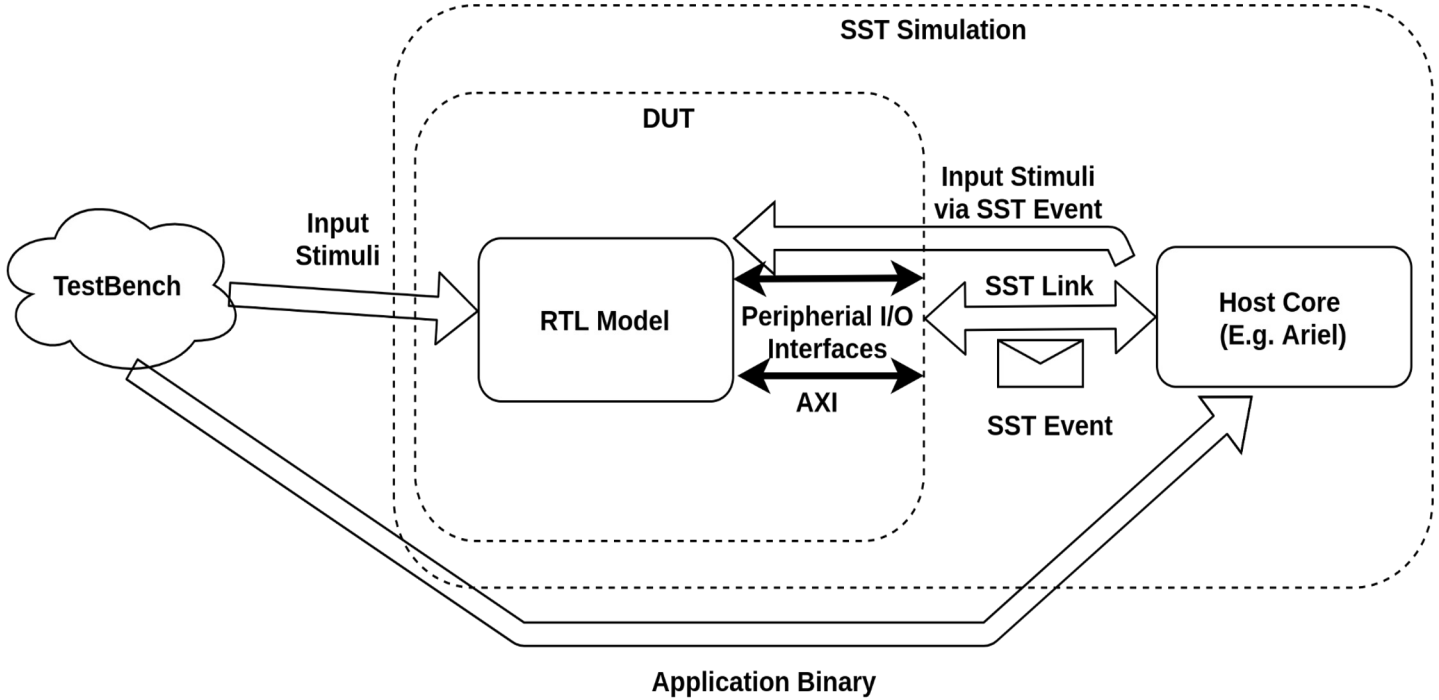
- Overview
- Motivation
- Framework Highlight
- **Parser**
- Handlers
- Summary
- Future Work

Parser Input Requirements

- Osseous APIs (Input Port spec and wrapper code spec)
  - Clock Handlers – Calls eval() that is clocked each cycle
  - memHandler – APIs generating & serving mem requests
  - Memory manager – Address translations
  - Event Handlers (e.g., interface handlers of AXI)
- Element information (SST_ELI_*)
  - SST Parameters, Statistics, additional ports, slots
  - Describe parameters from Python Config file
- Boilerplate code (init(), setup(), finish() APIs)
  - Initialize or load test vector/binary files
  - Creating testbench consumed as x86 binary



Input Stimulus



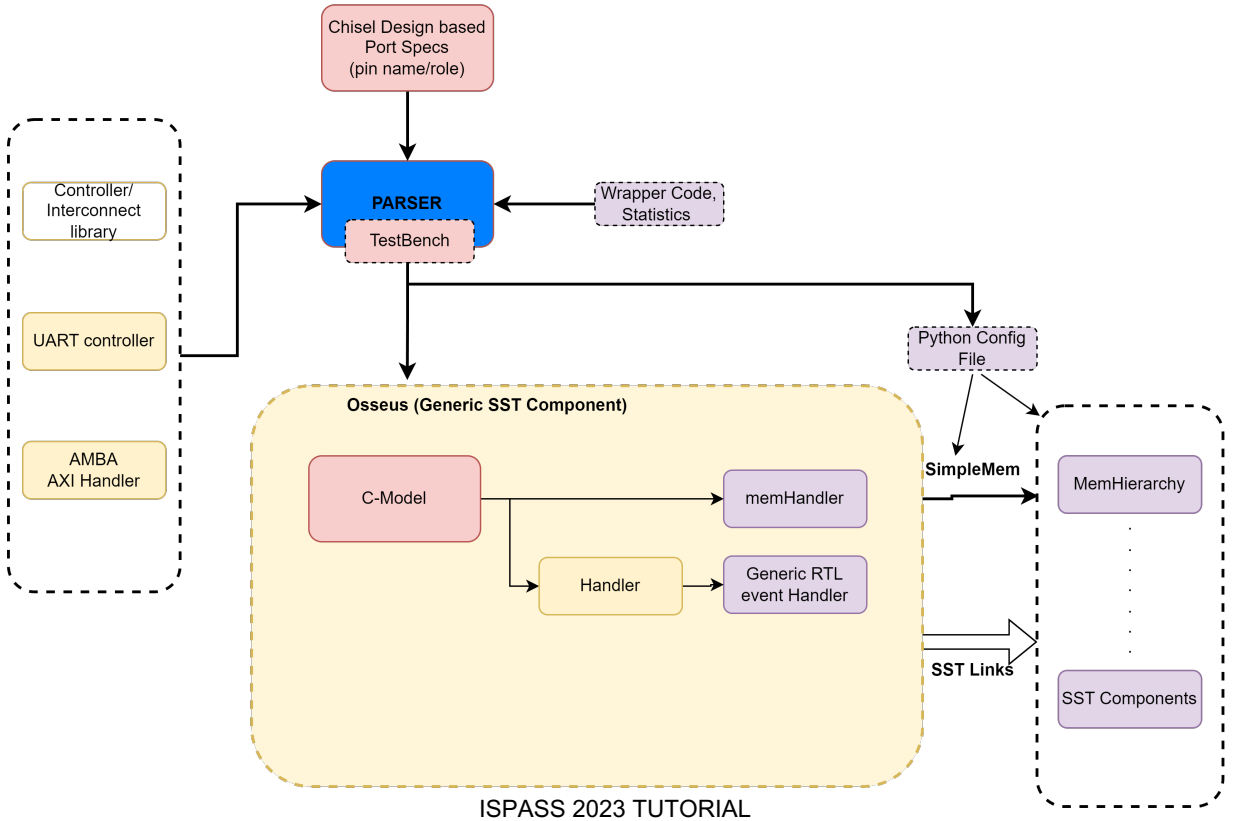
- Testbench/x86 binary or mem loadable Hex file
  - RTL Simulation requires input stimulus
  - TestBench to update RTL signals are written in C and compiled using gcc / g++
  - Custom APIs to offload computation to RTL model are created (e.g. start_RTL_sim(), update_RTL_sig())
  - Independent RTL requires loading Hex file to memHierarchy (e.g. RISC-V CPU)

Outline

- Overview
- Motivation
- Framework Highlight
- Parser
- **Handlers**
- Summary
- Future Work

Handlers

- Establish Link between SST world and C-model supported interfaces/ interconnect for inter-component communication
- Autogenerated code invoked by the parser on demand
- Handler records transaction statistics to ***evaluate performance impacts***



Outline

- Overview
- Motivation
- Framework Highlight
- Parser
- Handlers
- **Summary**
- Future Work

Summary

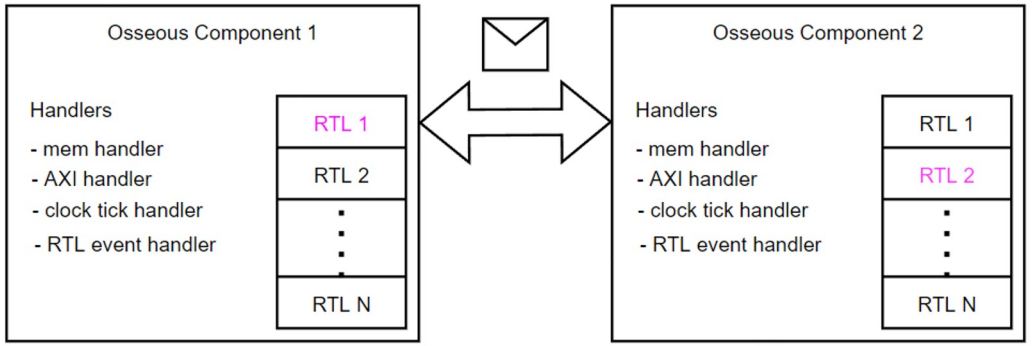
- Quick and seamless integration of low-level detailed simulation with highly modular, parallelizable architectural simulator such as SST facilitate:
  - Fast evaluation of performance impact of designed IP on abstract high level architecture models
  - System level study of detailed simulation to incorporate early design changes.
  - Design space exploration driven by real world IP constraints
  - Save significant efforts in modelling architectural components through exploiting reusability of existing models in SST

Outline

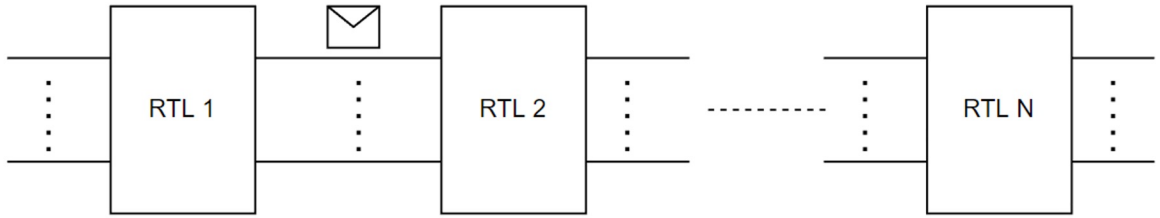
- Overview
- Motivation
- Framework Highlight
- Parser
- Handlers
- Summary
- **Future Work**

Future Work

- Instantiate and interface multiple Osseus components



- Take out the dependency on existing SST cores (e.g. Ariel) to run system-level simulation with Osseous



Demo

- **Example 1: Counter circuit**
  - Workflow and test setup
  - Verilog vs SST testbench
- **Example 2: RISCV-Mini**
  - AXI Handler
  - Design space exploration use case
- **Summary**

Example 1: Workflow and test setup

- The Verilog/Chisel model of the design is translated to FIRRTL which is then used to generate the C-model using Essent[1]
- The testbench is a C file which is compiled to a x86 binary and executed by Ariel CPU using PIN tool[2].
- The Ariel CPU writes the signal values received from the testbench to the Memhierarchy and notifies the DUT.
- The DUT fetches the latest signal values from the Memhierarchy and executes for cycles as determined by the testbench.

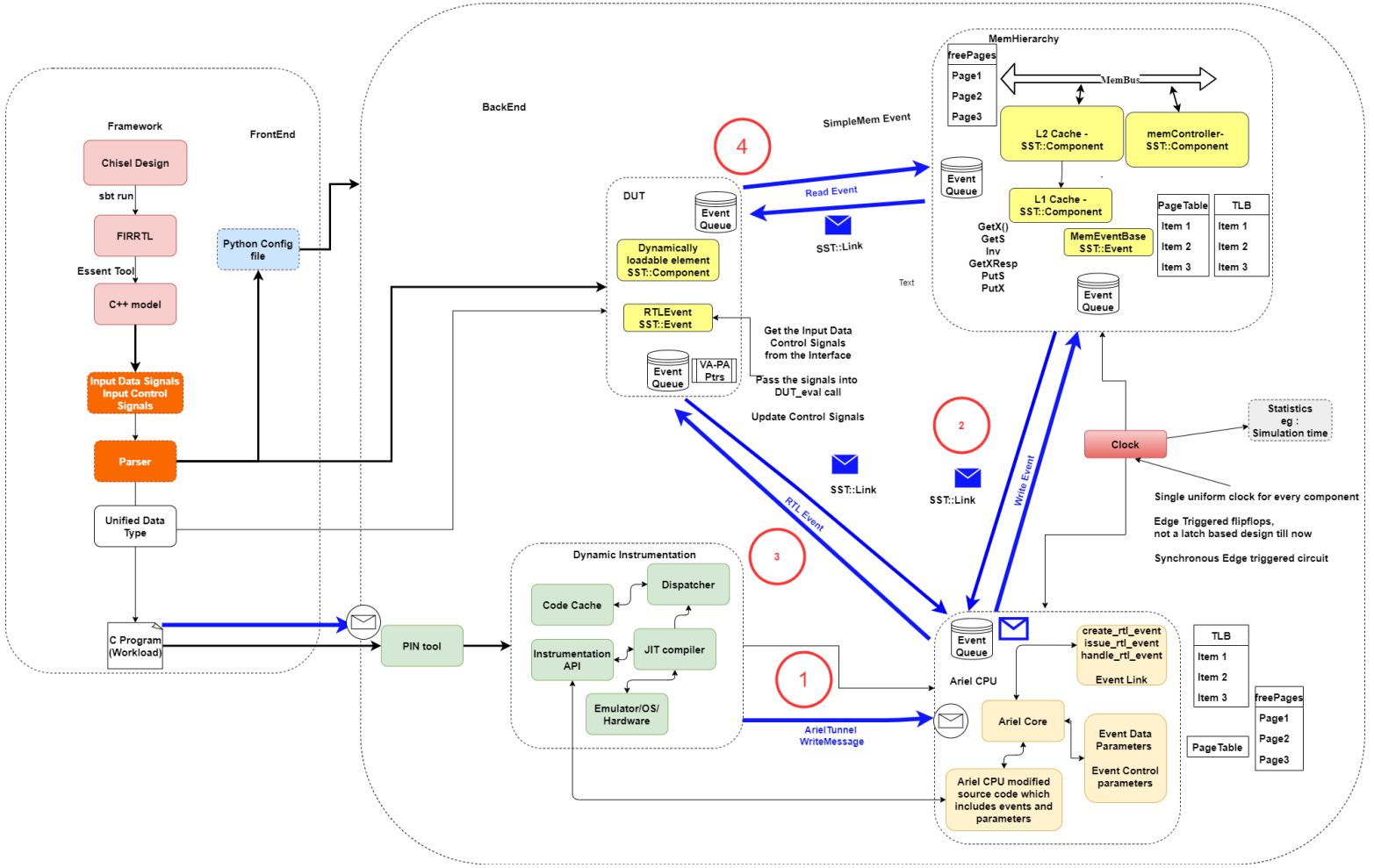


Figure 1: ERAS framework workflow and test setup

1: S. Beamer, "A Case for Accelerating Software RTL Simulation," in IEEE Micro, vol. 40, no. 4, pp. 112-119, 1 July-Aug. 2020, doi: 10.1109/MM.2020.2997639.

2: Pin - A Dynamic Binary Instrumentation Tool (intel.com)

Example 1: Counter circuit

```
package solutions
import chisel3._
object Counter {
  def wrapAround(n: UInt, max: UInt) =
    Mux(n > max, 0.U, n)
  def counter(max: UInt, en: Bool, amt: UInt): UInt = {
    val x = RegInit(0.U(max.getWidth.W))
    when (en) { x := wrapAround(x + amt, max) }
    x
  }
}
class Counter extends Module {
  val io = IO(new Bundle {
    val inc = Input(Bool())
    val amt = Input(UInt(4.W))
    val tot = Output(UInt(8.W))
  })
  io.tot := Counter.counter(255.U, io.inc, io.amt)
}
```

Figure 3: Chisel

```
module mCounter(clock, reset, io_inc,
                io_amt, io_tot);
  input clock;
  input reset;
  input io_inc;
  input [3:0] io_amt;
  output reg [7:0] io_tot;

  always @(posedge clock or
           negedge reset)
    begin
      if(!reset)
        begin
          io_tot <= 0;
        end
      else
        begin
          if(io_inc)
            begin
              io_tot<=io_tot+io_amt;
            end
        end
    end
endmodule
```

Figure 4: Verilog

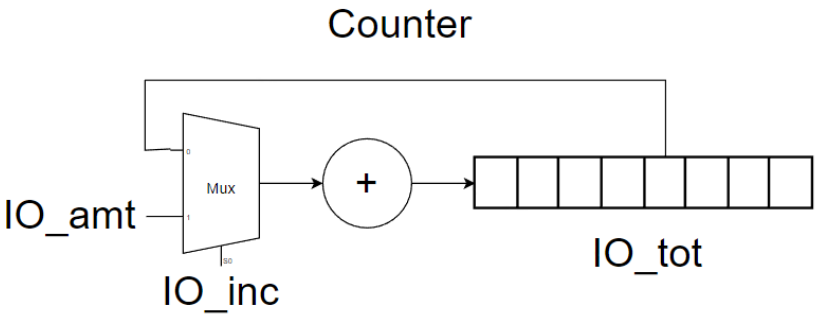


Figure 2: Circuit design

```
circuit Counter :
  module Counter :
    input clock : Clock
    input reset : UInt<1>
    output io : {flip inc : UInt<1>, flip amt : UInt<4>, tot : UInt<8>}

    reg _T : UInt<8>, clock with : (reset => (reset, UInt<8>("h00"))) @[Counter.scala 18:20]
    when io.inc : @[Counter.scala 19:15]
      node _T_1 = add(_T, io.amt) @[Counter.scala 19:35]
      node _T_2 = tail(_T_1, 1) @[Counter.scala 19:35]
      node _T_3 = gt(_T_2, UInt<8>("h0ff")) @[Counter.scala 15:11]
      node _T_4 = mux(_T_3, UInt<1>("h00"), _T_2) @[Counter.scala 15:8]
      _T <= _T_4 @[Counter.scala 19:19]
      skip @[Counter.scala 19:15]
    io.tot <= _T @[Counter.scala 32:10]
```

Figure 5: FIRRTL

Essent

```
void EVAL_0() {
  PARTflags[0] = false;
  io_tot = _T;
  UInt<8> _T$next;
  if (UNLIKELY(UNLIKELY(reset))) {
    _T$next = UInt<8>(0x0);
  } else {
    UInt<8> _GEN_0;
    if (UNLIKELY(io_inc)) {
      UInt<8> _GEN_1 = io_amt.pad<8>();
      UInt<9> _T_1 = _T + _GEN_1;
      UInt<8> _T_2 = _T_1.tail<1>();
      _GEN_0 = _T_2;
    } else {
      _GEN_0 = _T;
    }
    _T$next = _GEN_0;
  }
  PARTflags[0] |= _T != _T$next;
  if (update_registers) _T = _T$next;
}
```

Figure 6: C-model

Example 1: Test bench in Verilog and SST

```
reset = UInt<1>(1);
inp_ptr[0] = reset;

params->perform_update(true, true, true, true, \
                       true, false, false, 2);
params->storetomem(shmem);
inp_ptr[0] = reset;
params->storetomem(shmem);
update_RTL_sig(shmem);

reset = UInt<1>(0);
io_inc = UInt<1>(1);
io_amt= UInt<4>(1);
inp_ptr[0] = reset;
inp_ptr[1] = io_inc;
tmp_ptr[0] = io_amt;

params->perform_update(true, true, true, true, \
                       true, true, false, 4);
params->storetomem(shmem);
inp_ptr[0] = reset;
inp_ptr[1] = io_inc;
tmp_ptr[0] = io_amt;
params->storetomem(shmem);
update_RTL_sig(shmem);

io_amt = UInt<4>(2);
tmp_ptr[0] = io_amt;

params->perform_update(true, true, true, true, \
                       true, true, true, 4);
params->storetomem(shmem);
tmp_ptr[0] = io_amt;
params->storetomem(shmem);
update_RTL_sig(shmem);
```

Figure 7: SST testbench

```
module test_fixture;
parameter TC = 10;

reg clock;
reg reset, io_inc;
reg [3:0] io_amt;
wire [7:0] io_tot;

initial
begin
        $dumpvars;
        clock = 0;
        reset = 1;
        io_inc = 0;
        io_amt = 4'h1;
        #10 reset = 0;
        #10 reset = 1;
        #10 io_inc = 1;
        #40 io_inc = 0;
        #10 io_amt = 4'h2;
        #10 io_inc = 1;

        #40 $finish;
end

always #5 clock=~clock;  //10ns clock

  mCounter U1 (clock, reset, io_inc, \
               io_amt, io_tot);

endmodule /* test_fixture */
```

Figure 8: Verilog testbench

```
Rtlmodel-rtlaximodel-> Updated Rtl Params is: 1
Rtlmodel->sim cycles: 4
Rtlmodel->update inp: 1
Rtlmodel->update ctrl: 1
Rtlmodel->input sigs: 0x0<U1>Rtlmodel->input sigs: 0x1<U1>Rtlmodel->input sigs:
0x1<U4>
io tot: 0x00<U8>
io tot: 0x01<U8>
io tot: 0x02<U8>
Performing update on RTL params
Store to mem called
Store to mem finished
io tot: 0x03<U8>ArielComponent[arielcore.cc:1539:processNextEvent]
ArielRTLEvent is being issuedRtlmodel-rtlaximodel->
VecshiftReg RTL Event handle called
Rtlmodel-rtlaximodel->
Rtlmodel-rtlaximodel-> Updated Rtl Params is: 1
Rtlmodel->sim cycles: 4
Rtlmodel->update inp: 1
Rtlmodel->update ctrl: 1
Rtlmodel->input sigs: 0x0<U1>Rtlmodel->input sigs: 0x1<U1>Rtlmodel->input sigs:
0x2<U4>
io tot: 0x04<U8>
io tot: 0x06<U8>
io tot: 0x08<U8>
io tot: 0x0a<U8>Rtlmodel-rtlaximodel-> OKToEndSim, TickCount
4ArielComponent[arielcore.cc:1400:handleRtlAckEvent]
Ariel received Event from RTL. Generating Read Request
```

Figure 9: SST test log

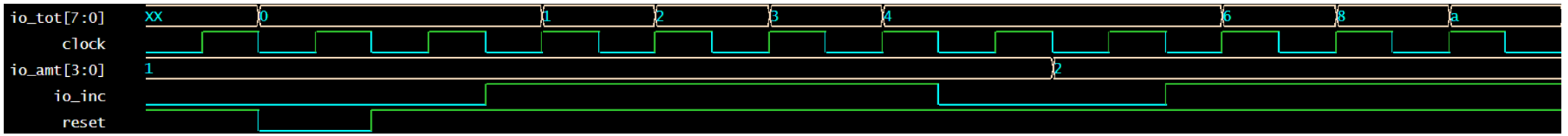


Figure 10: Verilog test output

Example 2: Riscv-Mini & AXI handler

- Riscv-Mini[3] is a RISC-V 3-stage pipeline written in Chisel
- The package comes with a simple inbuilt memory manager. It loads the binary to the memory which has a fixed access latency of 1 cycle
- In this example, we replace the in-built memory manager with SST's memhierarchy model
- The Riscv-Mini CPU interfaces with memory manager using AXI4. However, SST's memhierarchy model doesn't have an inbuilt AXI interface

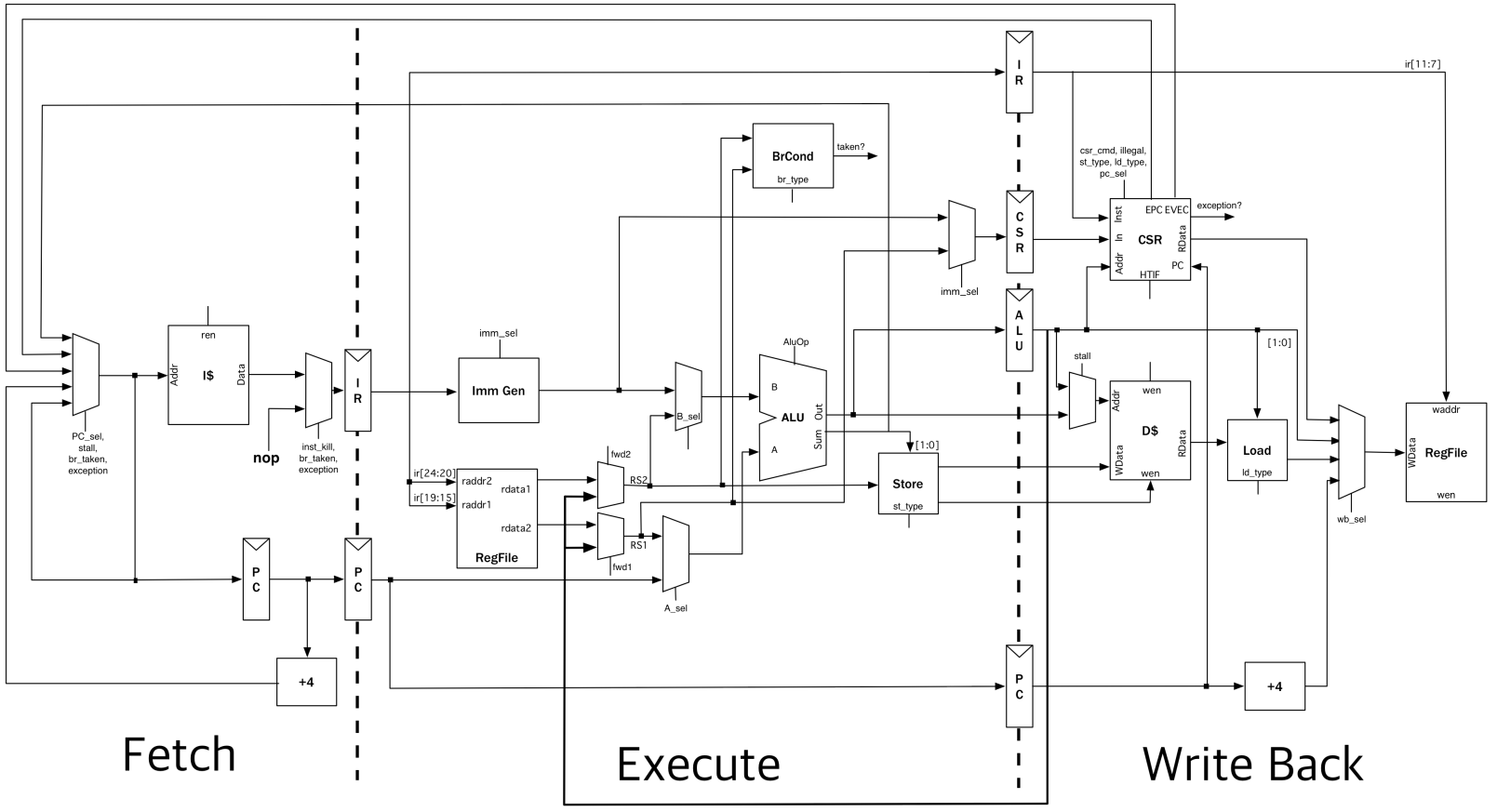


Figure 11: Riscv-Mini microarchitecture

3: https://github.com/ucb-bar/riscv-mini

Example 2: Riscv-Mini & AXI handler (cont'd)

- The AXI handler acts as a slave device which translates read/write requests received over AXI bus to SST events to read/write data from the memhierarchy.
- The AXI handler synchronizes the updating of bus signals and communicating SST events.
- A benchmark (quick sort) is loaded to memhierarchy and executed on the Riscv-Mini CPU to analyze the IPC for different L1 cache configurations

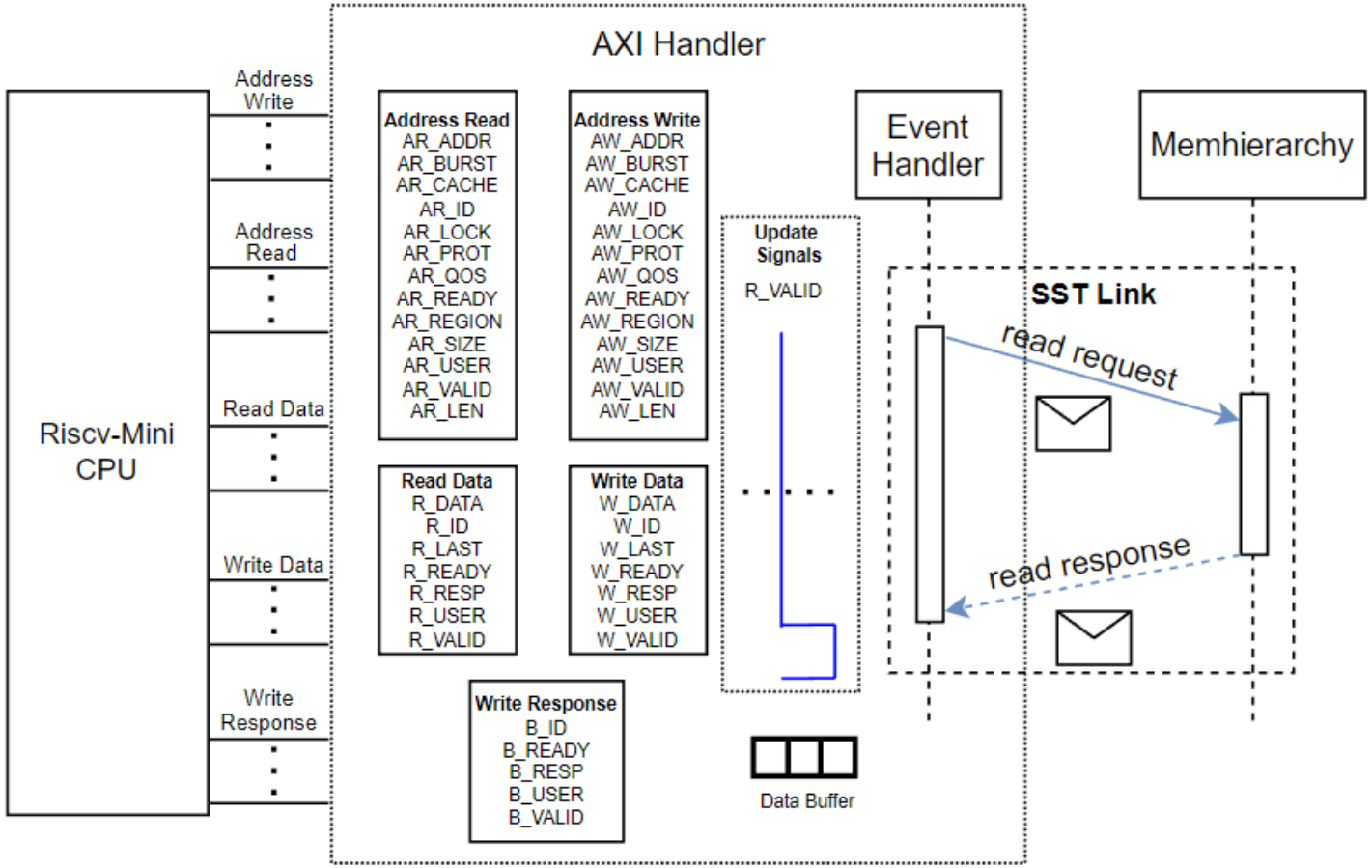


Figure 12: AXI Handler

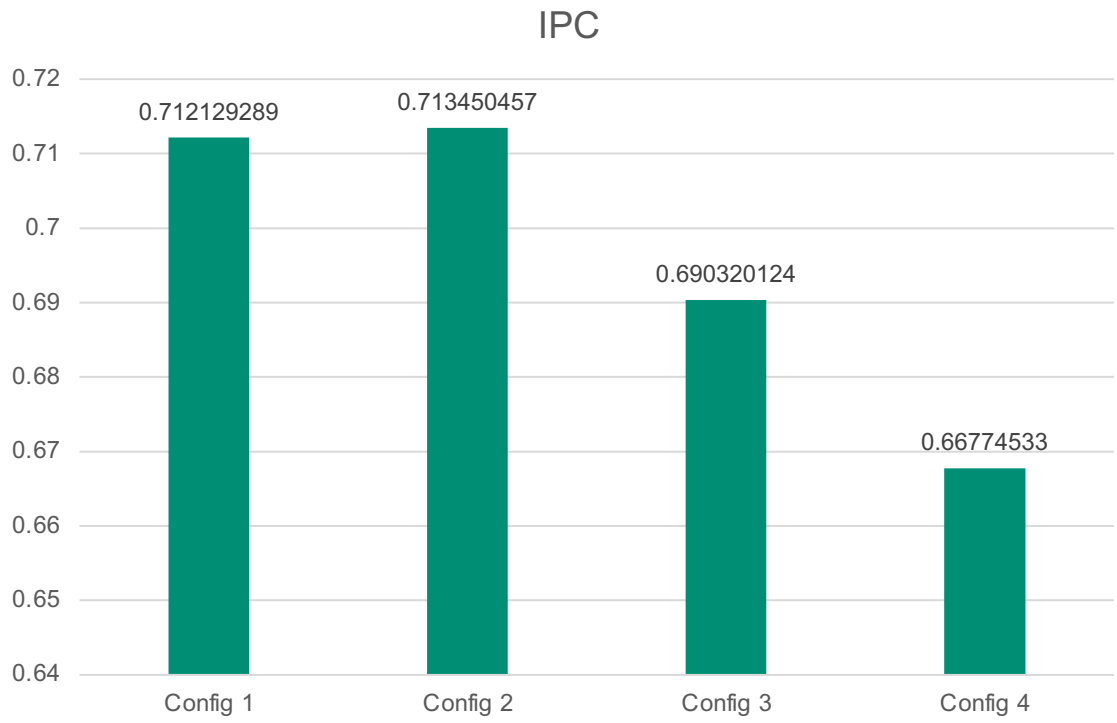
Design space exploration of L1 cache with Riscv-Mini CPU RTL

- **Config 1**
  - cache size : 2 KB
  - cache type : inclusive
  - coherence protocol : MSI
  - replacement policy : lru
  - associativity : 2
  - access latency cycles : 1
  - cache line size : 64

- **Config 2**
  - cache size : 32 KB
  - cache type : inclusive
  - coherence protocol : MSI
  - replacement policy : lru
  - associativity : 4
  - access latency cycles : 4
  - cache line size : 64

- **Config 3**
  - cache size : 64 KB
  - cache type : inclusive
  - coherence protocol : MSI
  - replacement policy : lru
  - associativity : 8
  - access latency cycles : 6
  - cache line size : 64

- **Config 4**
  - cache size : 128 KB
  - cache type : inclusive
  - coherence protocol : MSI
  - replacement policy : lru
  - associativity : 16
  - access latency cycles : 8
  - cache line size : 64



Contributors

- Sandia National Laboratories
  - C. Hughes, A. Rodrigues, G. Voskuilen, K. S. Hemmert, S. D. Hammond, B. Feinberg
    Center for Computing Research
    {chughes, afrodi, grvosku, kshemme, sdhammo, bfeinbe}@sandia.gov

- North Carolina State University
  - S. K. Chunduru, S. Nema, M. Ambati, R. Razdan, J. Kirschner, D. Adak, H. Lee, A. Awad
    Secure and Advanced Computer Architecture Lab
    {schundu2, snema, mambati, rrazdan2, dadak, hyokeun_lee, jkirsch, ajawad}@ncsu.edu

Thank You!

Slide Left Blank