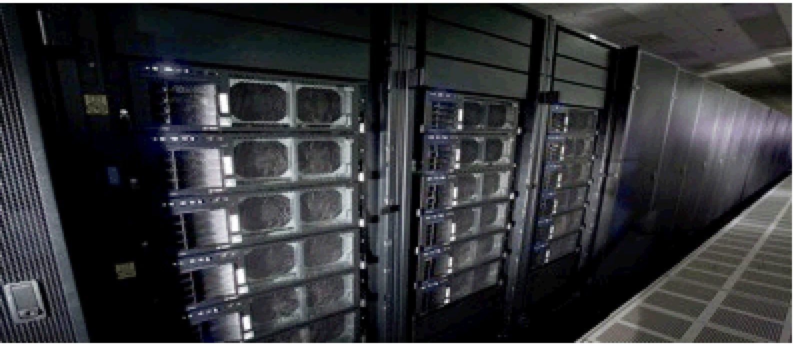
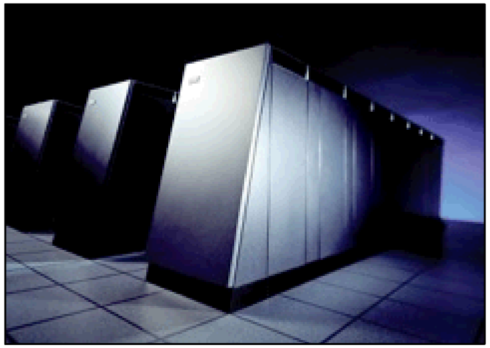
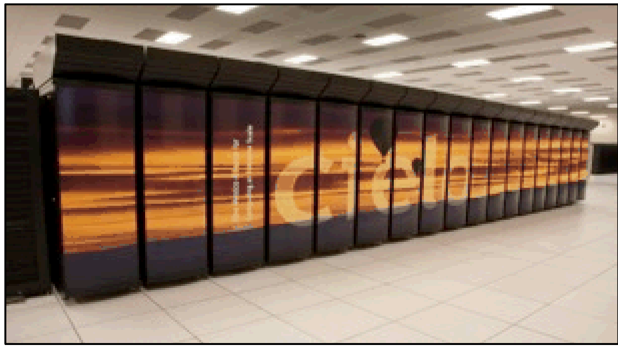
SAND2017-7486PE







SST Tutorial – “Juno” Example Processor

SST Development Team

Sandia National Laboratories, NM

sst@sandia.gov

CCR
Center for Computing Research

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and 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.

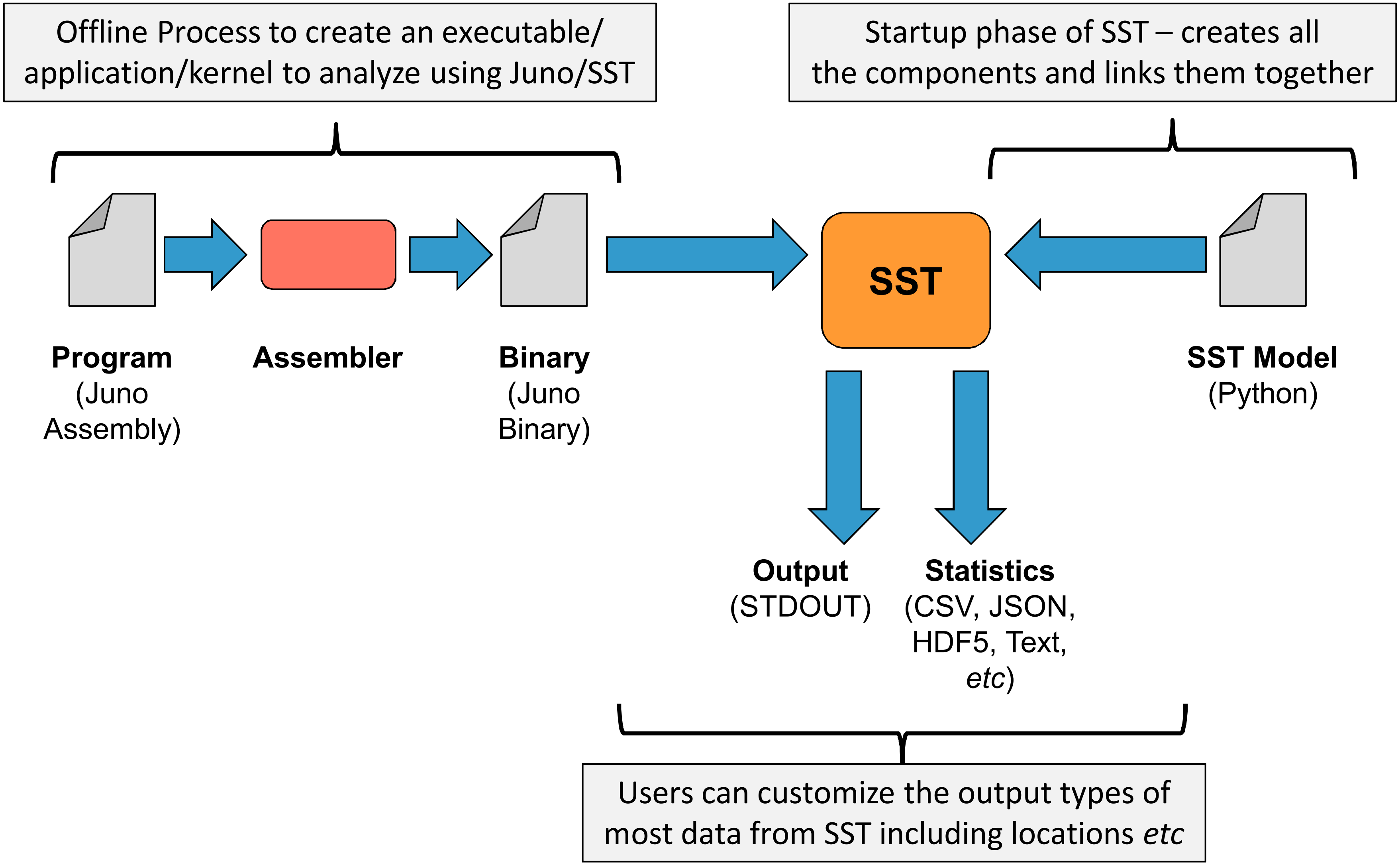
Welcome to the SST Juno Tutorial!

- **Tutorial Goal:** become familiar with API, structures and design patterns for building components and subcomponents in SST

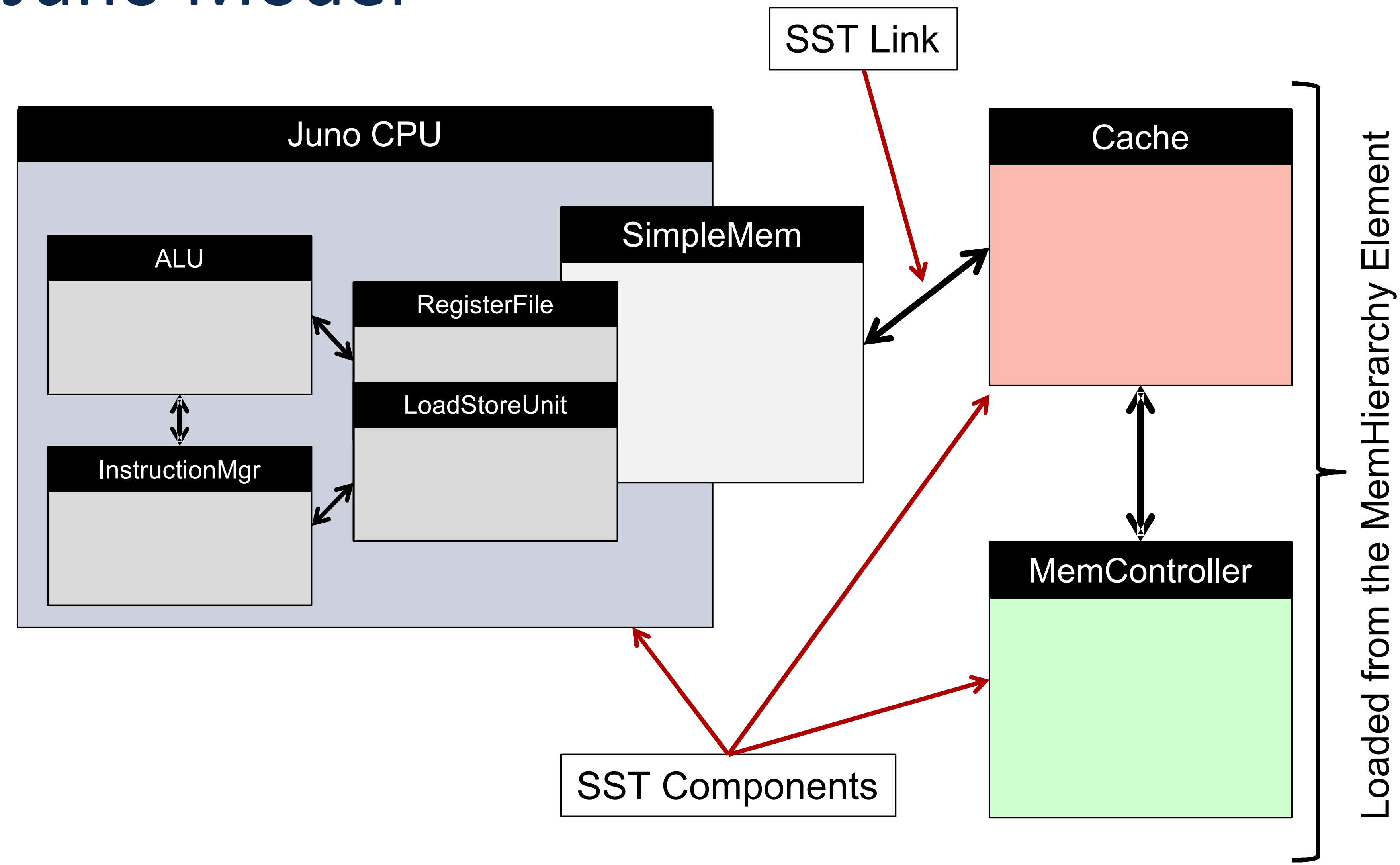
- Juno is a (very simple) example cycle-approximate execution-driven processor core
  - Operates on 64-bit signed integers only
  - Very limited number of built-in instructions
    - But extensible with custom instructions (we will show how)
  - Interfaces with SST memory sub-system models
  - Utilizes many of the basic structures offered by SST to make developing architectural simulation models much easier

https://github.com/sstsimulator/juno

Basic Juno Model/Workflow



Juno Model


SST Link
Juno CPU
ALU
RegisterFile
LoadStoreUnit
InstructionMgr
SimpleMem
Cache
MemController
SST Components
Loaded from the MemHierarchy Element

Tutorial Outline

- There are five exercises to introduce you to the parts of SST

- **Exercise 1** – get a basic Juno model running using SST's Python model scripts
- **Exercise 2** – add a configuration parameter to Juno to change the execution behavior
- **Exercise 3** – add statistics support to Juno to track metrics of interest during simulation execution
- **Exercise 4** – add a new instruction to Juno using SST's SubComponent interface
- **Exercise 5** – add an external "accelerator" to Juno to demonstrate inter-component connectivity

Exercise 1 – Use SST to Run a Juno Program

Exercise 1 – Running a Juno App

- **Goal:** The first exercise is to use SST to run a Juno program

- **Hint:** sst ./juno-exercise-001.py

- **Extra: Change the Program Being Run:**
  - Second Juno program (isqrt.juno) needs to be assembled
  - Edit juno-exercise-001.py
  - Change the application being run and repeat

- **Extra: Change the verbosity of the CPU model**
  - Change the verbose parameter to 1, 2, 4, 8, .. 32 and re-run

Exercise 2 – Adding a Parameter to the Juno Model

Exercise 2 – Adding Parameters

- **Goal**: Add a parameter to Juno to control its behavior
- **Description:** (1) parameters require definition in the "manifest" so that SST can check we are loading the right values; (2) we can use the parameter
- **Activity:** add the "clock" parameter to control the simulated clock rate of Juno
- **Hint:** look at the `SST_ELI_DOCUMENT_PARAMS` macro in junocpu.h
- **Check**: add the "clock" parameter in juno-exercise-002.py, try different values and re-run

Exercise 3 – Add Metrics ("Statistics") to Juno Model

Exercise 3 - Statistics

- **Goal** – add statistics capture into the Juno model to allow users to see behavior

- **Description:** (1) Statistics must also be registered in the manifest for the model; (2) Statistics must then be created in the component; (3) Statistics can have data added to them during execution

- **Hint (1):** Look at the `SST_ELI_DOCUMENT_STATISTICS` in junocpu.h (this registered statistics values)

Exercise 3 - Statistics

- To use statistics from the SST core you need to use the following:

- In your model class add a member:
  - `Statistic<uint64_t>* statCycles;`
  - (Creates a unsigned 64-bit integer statistic value for use as a metric)

- In your model constructor:
  - `statCycles = registerStatistic<uint64_t>( "cycles" );`
  - Registers the statistic with the core so it can be incorporated into the unified output

- In the code which runs your model:
  - `statCycles->addData(1);`

Exercise 3 –Statistics and Python

- Once your model has statistics enabled, we must tell SST which ones to turn on during execution (so we are not overwhelmed)

- At the end of juno-exercise-003.py (create a CSV dump)

```
# Set the statistics to output
sst.setStatisticOutput("sst.statOutputCSV")
sst.enableAllStatisticsForAllComponents()

sst.setStatisticOutputOptions( {
        "filepath"  : "output.csv"
} )
```

Exercise 4 – Adding a New Instruction using SubComponents

Exercise 4 – Add a SubComponent

- SubComponents are sub-parts of a full component which can be dynamically loaded into a model. In this case Juno has several built-in instructions but can also load in additional user-defined extensions
- **Goal**: load a new instruction subcomponent into the Juno model so we can add RAND and RSEED instruction support (assembler has already been modified to generate RAND and RSEED output)
- **Description**: a random instruction sub-component has already been developed (see src/custominst/junorandinst.h)
- **Activity**: (1) add a subcomponent "slot" into Juno; (2) modify juno-exercise-004.py so we can load the subcomponent into the Juno CPU and then run a simple GUPS program

Exercise 4 – SubComponent Slot

- SubComponent "slots" tell SST that it should expect to load a new additional piece of code into this space

- In junocpu.h we need to add the following:

```
SST_ELI_DOCUMENT_SUBCOMPONENT_SLOTS(
      {"customhandler", "Holds customer instruction handlers",
                        "SST::Juno::CustomInstructionHandler" }
      )
```

- This defines a slot called "customhandler" (customhandlers in Juno handle instructions not matched by the processors default ISA)

Exercise 4 – Random SubComponent

- In juno-exercise-004.py add the following:

```
# Define RAND support
randsc = comp_cpu.setSubComponent("customhandler",
"juno.JunoRandomHandler")
randsc.addParam("seed", 131313)
```

- This tells SST you want to load an instance of `juno.JunoRandomHandler` into the "customhandler" slot we just defined

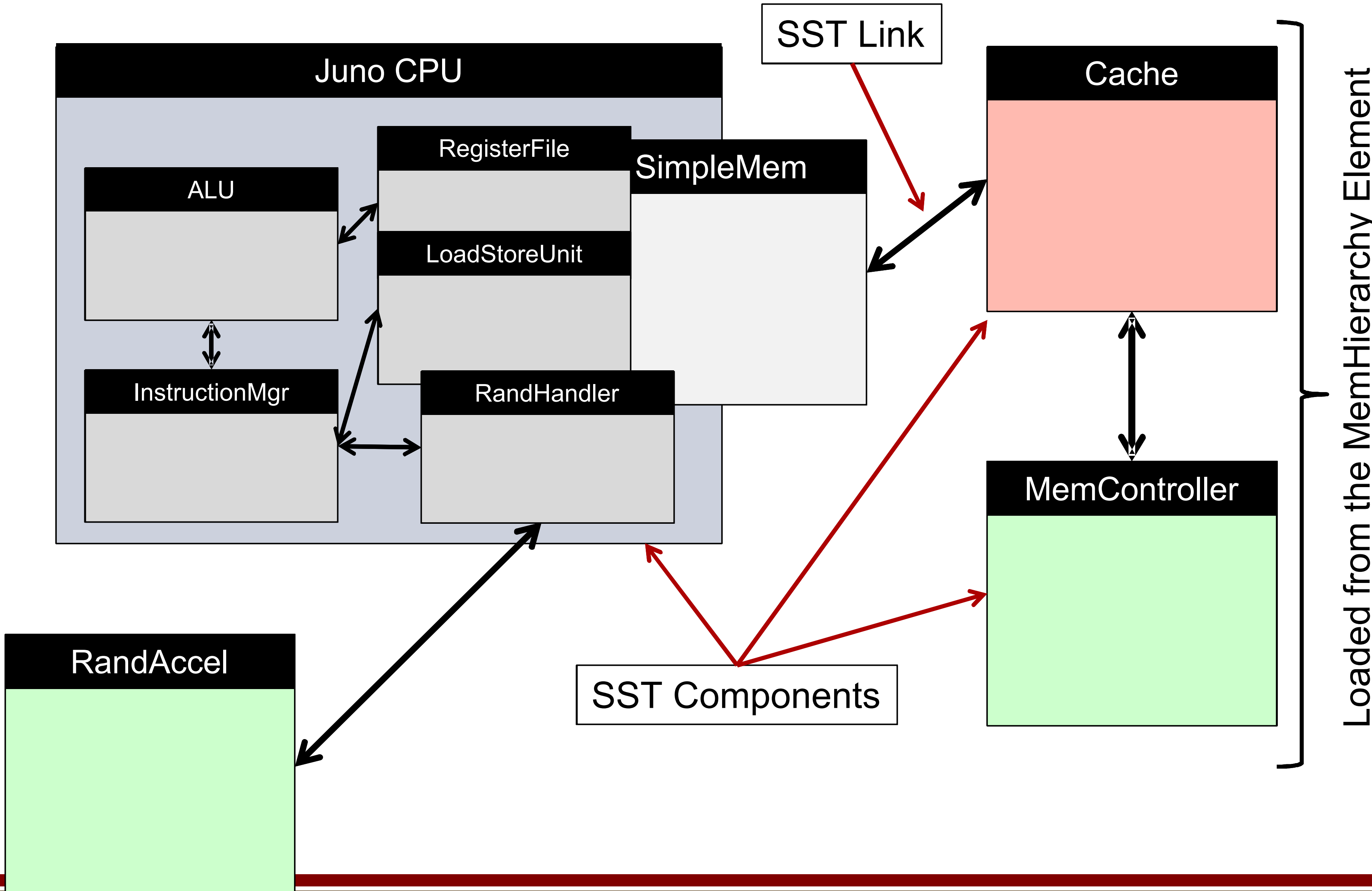
- **Activity**: compile GUPS and use SST to run juno-exercise-004.py

Exercise 5 – Use External Components to Provide Random Support

Exercise 5 – Connect Components

- Imagine that our architecture department has developed an external “random number” accelerator we want to attach to the Juno CPU
- **Goal**: attach an external “random accelerator” component to Juno to create random numbers for our applications
- **Activity:** (1) we need to create a new component; (2) we need to create a subcomponent (“customhandler” for Juno) which can connect externally; (3) we need to create a link between them

Exercise 5- External Connectivity



Exercise 5 – External Connectivity

- **Step 1** – create a new component in juno-exercise-005.py:

```
# Define external RAND accelerator
rand_accel = sst.Component("randacc",
"juno.JunoRandAccelerator")
rand_accel.addParams({
    "verbose" : 1
})
```

- Creates a new component called “randacc” which is already written and supplied by Juno’s element library

Exercise 5 – External Connectivity

- **Step 2 –** we need to create a new ExternalRandomHandler for Juno (routes RAND instructions off the CPU and manages the connection)

```
# Define RAND support
randsc = comp_cpu.setSubComponent("customhandler",
"juno.JunoExternalRandomHandler")
```

- This is a subcomponent of the Juno CPU because this provides the connection from Juno to the new component

Exercise 5 – External Connectivity

- **Step 3 –** connect the new random accelerator component to the Juno random handler

```
cpu_rand_link = sst.Link("cpu_rand_accel_link")
cpu_rand_link.connect(  (randsc, "genlink",
"2ns"), (rand_accel, "cpulink", "2ns") )
```

- “genlink” and “cpulink” are named ports in the element manifest (so SST knows how to connect everything together)

Exercise 5 – Run!

- **Step 4 –** run juno-exercise-005.py
- **Extra** – you can turn up the verbose settings on the components to see more information get printed about the messages between them
- **Extra** – change the parameters in the Python script and see what happens to the projected performance