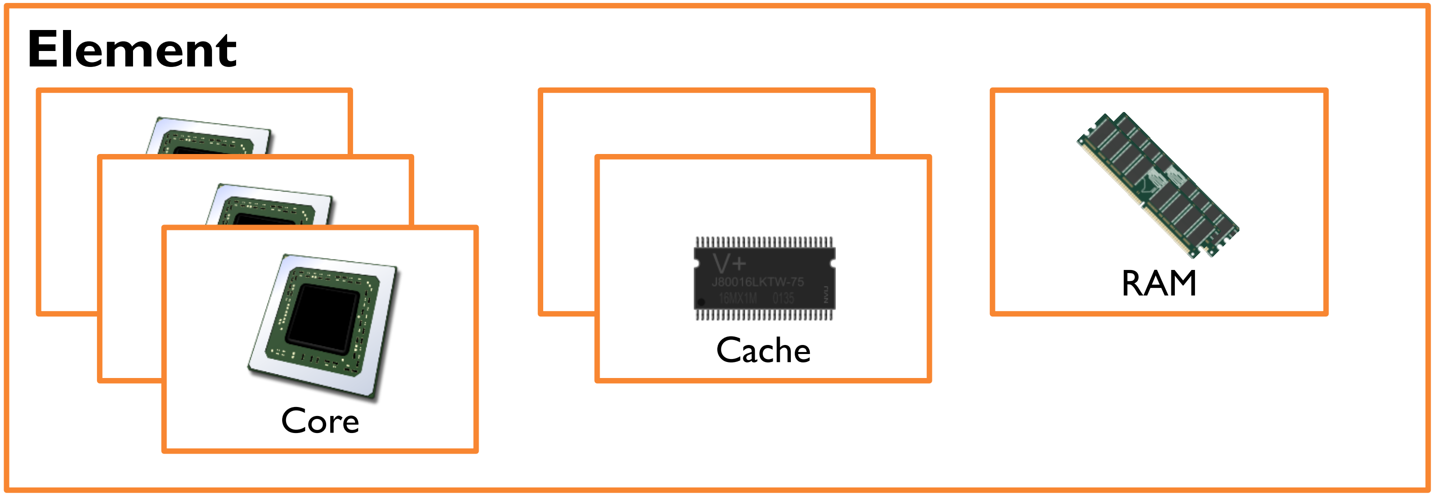
A Brief introduction to SST

SST simulators

Structural Simulation Toolkit

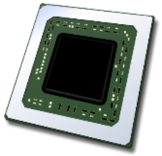
- Is a framework for event-driven simulation
  - Specializes in architecture simulation
  - Can be used for other applications
- Is composed of **elements**
  - **Libraries** with simulators and other tools



SST simulators

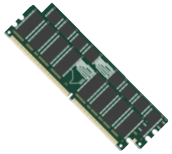
Components

- **Components** are building block of simulations
  - Performs the actual simulation
  - Can be processors, cache, memory, etc.



Core

Cache

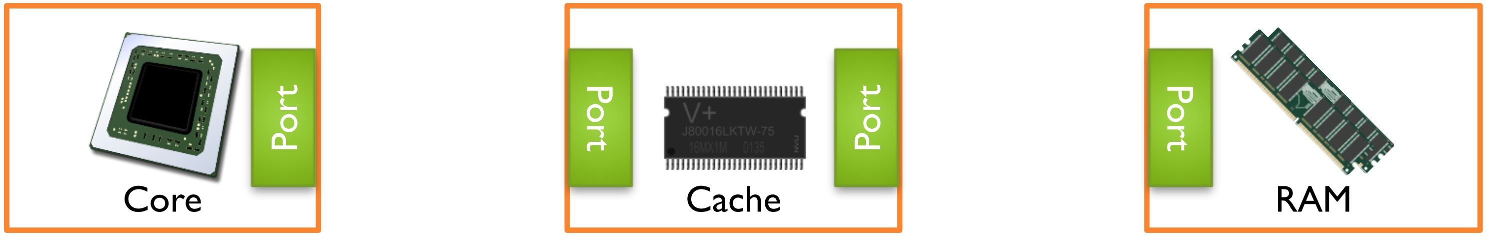


RAM

SST simulators

Components

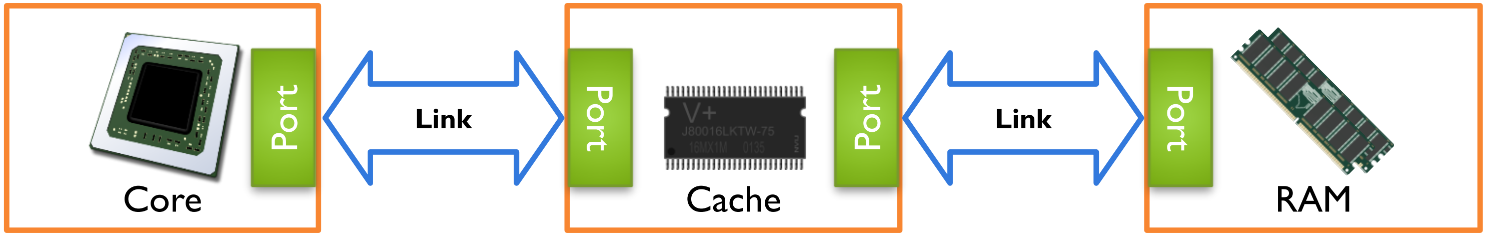
- **Components** are building block of simulations
  - Performs the actual simulation
  - Can be processors, cache, memory, etc.
- Components have **ports**
  - Used to communicate information between them



SST simulators

Links

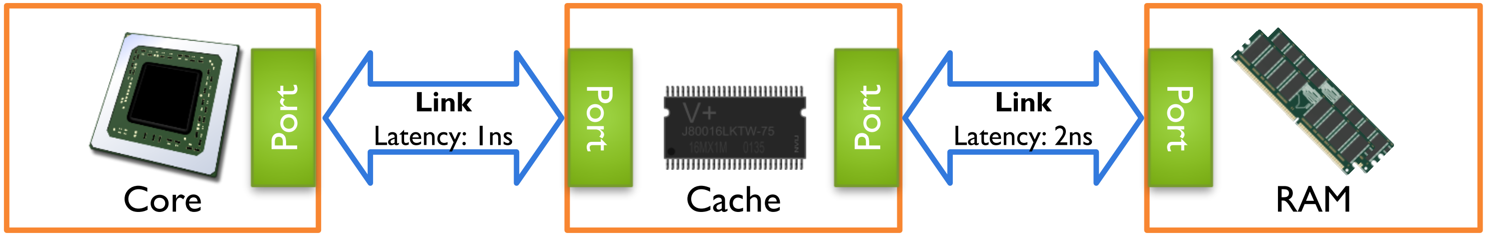
- **Links** connect components
  - **One-to-one** connections between two ports
  - Unique form of communication



SST simulators

Links

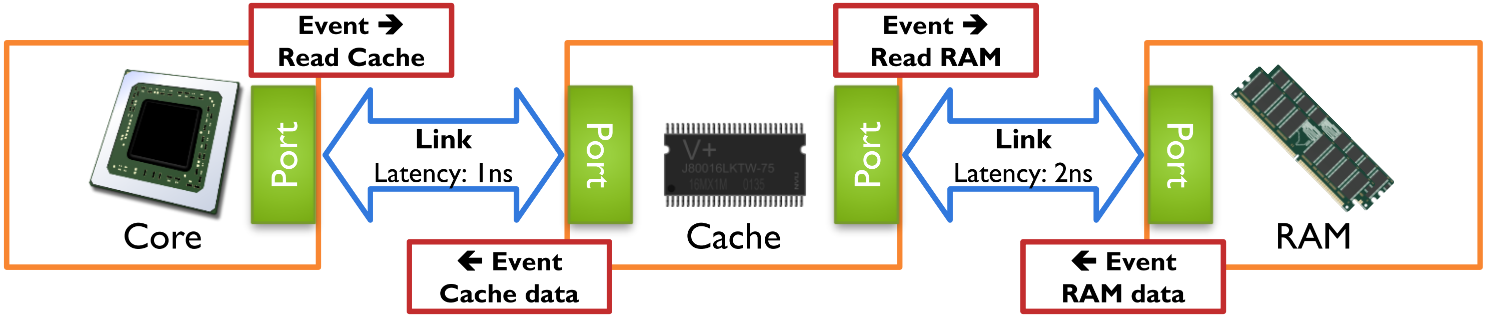
- **Links** connect components
  - **One-to-one** connections between two ports
  - Unique form of communication
- Have **non-zero latency**
  - Except for some special cases (self-links)



SST simulators

Events

- **Comm. between** two **components**
  - Go through the links
  - No predefined format! (**Flexible**)

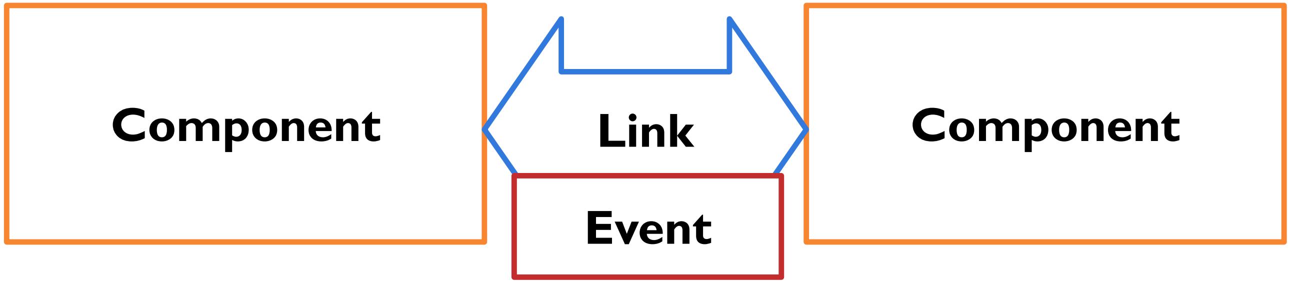


Assembling simulations

SST simulators

Summing up

- Simulations are composed of **components**
- **Links** connect the components
- **Components** send **events** through the links



SST simulations

Configuration

- SST uses a Python configuration file
  - Defines global parameters for the simulation
  - Defines and configures components
  - Specifies links and link latencies between components

SST simulations

Configuration

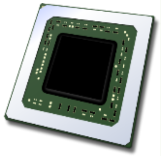
- **Define:** `sst.Component(“name”, “type”)`
- **Configure:**
  - `addParams({ “parameter” : value, … })`

```
import sst

core = sst.Component("XSim","XSim.core")

core.addParams({
     "clock_frequency": “1GHz”,
     "program": args.program,
     "verbose": 0
})

core.addParams(latencies)
```



SST simulations

Configuration

- **Define:** `sst.Component(“name”, “type”)`
- **Configure:**
  - `addParams({ “parameter” : value, … })`

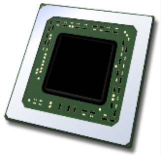
```
import sst

core = sst.Component("XSim","XSim.core")

core.addParams({
    "clock_frequency": “1GHz”,
    "program": args.program,
    "verbose": 0
})

core.addParams(latencies)
```

```
latencies={
  “liz”: 10,
  “sw”: 100,
  “lw”:100
}
```



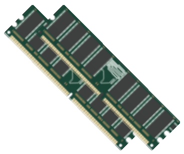
SST simulations

Configuration

- **Define:** `sst.Component(“name”, “type”)`
- **Configure:**
    - `addParams({ “parameter” : value, … })`

```
memory = sst.Component("data_memory",
                        "memHierarchy.MemController")

memory.addParams({
    'clock’: “1GHz",
    'backend':"memHierarchy.simpleMem",
    'backend.mem_size’: "64KiB",
    'backend.access_time’: “100ns“
})
```



SST simulations

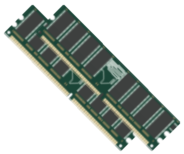
Configuration

- **Define:** sst.Component(“name”, “type”)
- **Configure:**
  - addParams({ “parameter” : value, … })

Where do these parameters come from?

```
memory = sst.Component("d
                       "memHierarchy.MemController")

memory.addParams({
    'clock’: “1GHz",
    'backend':"memHierarchy.simpleMem",
    'backend.mem_size’: "64KiB",
    'backend.access_time’: “100ns“
})
```



SST simulations

Configuration

- **Define:** sst
- **Configure**
  - addParams({

Try running on the command line:

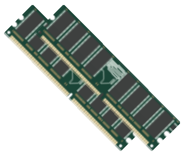
```
sst-info memHierarchy.MemController
```

And:

```
sst-info memHierarchy.simpleMem
```

```
memory = sst.Component("data_memory",
                       "memHierarchy.MemController")

memory.addParams({
    'clock’: “1GHz",
    'backend':"memHierarchy.simpleMem",
    'backend.mem_size’: "64KiB",
    'backend.access_time’: “100ns“
})
```



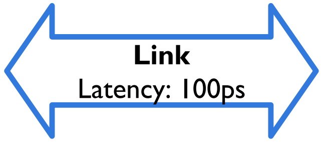
SST simulations

Configuration

- **Create a link:** `sst.Link(“name”)`
- **Connect link endpoints:**
  - `connect(endpoint1, endpoint2)`
    - Endpoint is defined as: (Component, Port, Latency)

Link name

```
cpu_data_memory_link = sst.Link("cpu_data_memory_link")
cpu_data_memory_link.connect(
  (core, "data_memory_link", "100ps"),
  (memory, "direct_link", "100ps")
)
```



Components

Port names

Go to code

SST simulations

Running the sst simulation

- **Running the sst simulation:**

```
On the command line run:
sst --add-lib-path build config.py
    --
    --program program.m
    --latencies latencies.json
```

SST simulations

Running the sst simulation

- **Running the sst simulation:**

```
On the command line run:
sst --add-lib-path build config.py
    --
    --program program.m
    --latencies latencies.json
```

```
--add-lib-path <path>
This is used to let SST know
where to find elements that
are not part of SST
(E.g. the CPU you developed)
```

SST simulations

Running the sst simulation

- **Running the sst simulation:**

```
On the command line run:
sst --add-lib-path build config.py
    --
    --program program.m
    --latencies latencies.json
```

config.py

This is the configuration file where you define the simulation

SST simulations

Running the sst simulation

- **Running the sst simulation:**

```
On the command line run:
sst --add-lib-path build config.py
    --
    --program program.m
    --latencies latencies.json
```

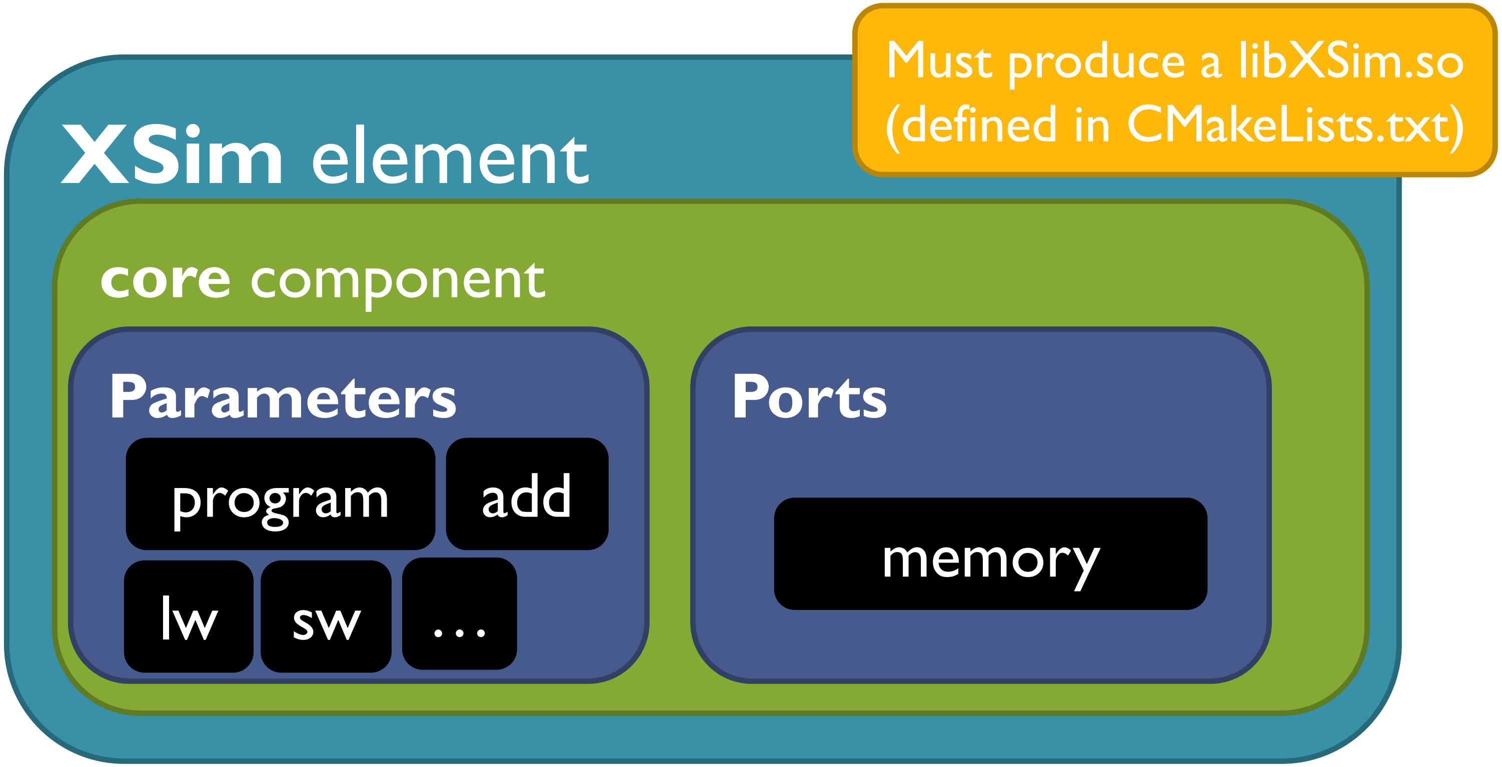
What comes after “--“ is passed directly to your script

Creating an element

XSim element

The xsim_sst_interface

- **Defined in file: xsim_sst_interface.cpp**



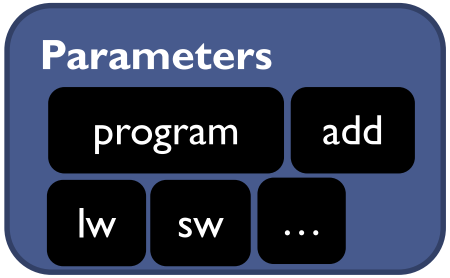
Implementing the core

CPU simulation

The core

- **Read configuration parameters**

```
params.find<std::string>(“frequency”,<default>)
load_latencies()
```



- **Load program**

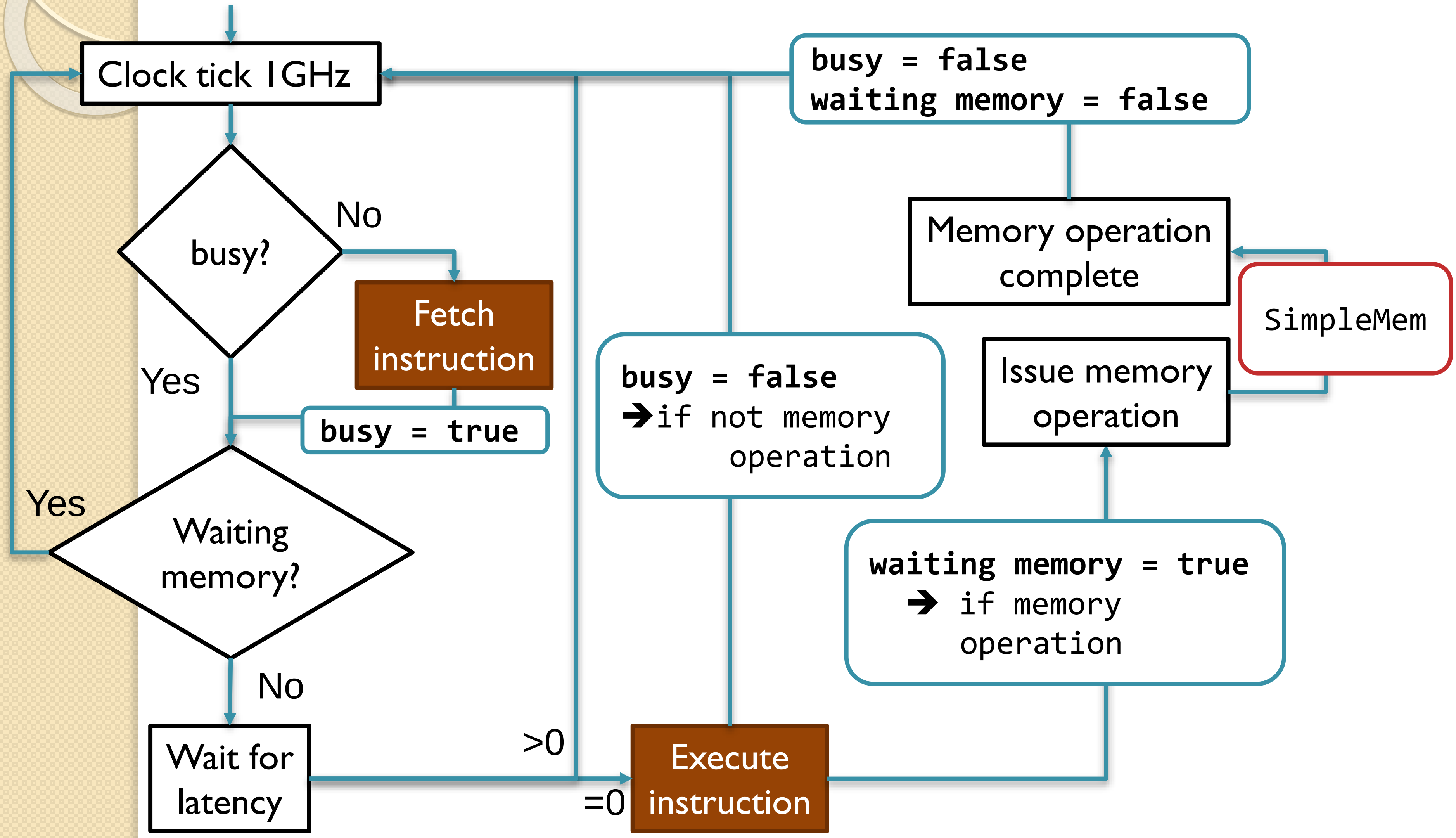
```
Std::vector<uint16_t> program ←
```

```
801F
8304
4860
4360
7060
6800
```

- **Start clock tick**

The CPU simulator

Clock-based simulation



CPU simulation

The core

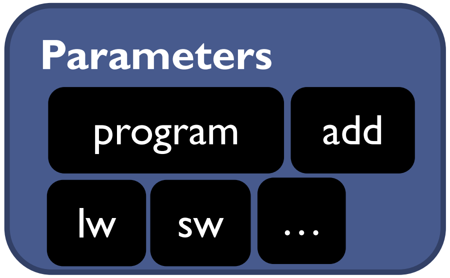
- **Read configuration parameters**

```
params.find<std::string>("frequency",<default>)
load_latencies()
```

- **Load program**

```
Std::vector<uint16_t> program ←
```

```
801F
8304
4860
4360
7060
6800
```


Parameters

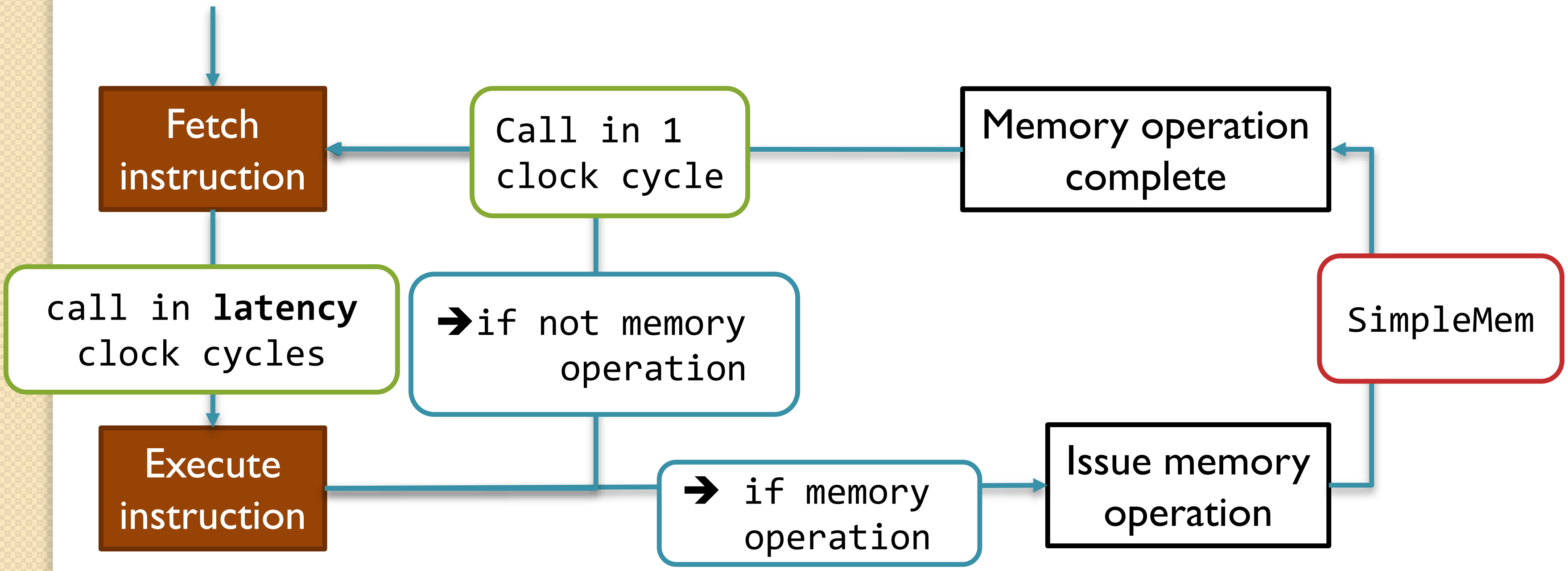
program | add

lw | sw | …

- **Send first fetch event**

The CPU simulator

Event-based simulation



Fetch instruction

Call in 1 clock cycle

Memory operation complete

call in **latency** clock cycles

➔if not memory operation

SimpleMem

Execute instruction

➔ if memory operation

Issue memory operation

Interfacing the memory

CPU simulation

The memory interface

- **Setup CPU memory connection**

```
data_memory_link = dynamic_cast<SimpleMem*>(
    … “memHierarchy.memInterface” …);
```

```
SimpleMem::Handler<Class> *data_handler=
  new SimpleMem::Handler<Class>(this, &Class::function);
```

```
data_memory_link.initialize(“data_memory_link”,
                            data_handler)
```

CPU simulation

The memory interface

- **Setup CPU memory connection**

```
data_memory_link = dynamic_cast<SimpleMem*>(
     … “memHierarchy.memInterface” …);
```

SST interface to talk with the memory controller

```
SimpleMem::Handler<Class> *data_hand
  new SimpleMem::Handler<Class>(this
```

```
data_memory_link.initialize(“data_memory_link”,
                                   data_handler)
```

CPU simulation

The memory interface

- **Setup CPU memory connection**

```
data_memory_link = dynamic_cast<SimpleMem*>(
    … “memHierarchy.memInterface” …);
```

```
SimpleMem::Handler<Class> *data_handler=
  new SimpleMem::Handler<Class>(this, &Class::function);
```

Wrapper for the function that handles completion events

```
data_memory_link.initialize(“data_
                            data_handler)
```

CPU simulation

The memory interface

- **Setup CPU memory connection**

```
data_memory_link = dynamic_cast<SimpleMem*>(
    … “memHierarchy.memInterface” …);
```

```
SimpleMem::Handler<Class> *data_handler=
  new SimpleMem::Handler<Class>(this, &Class::function);
```

```
data_memory_link.initialize(“data_memory_link”,
                            data_handler)
```

Connect **port** and wrapper

CPU simulation

The memory interface

Memory values are not stored by SimpleMem!
You have to handle that yourself.

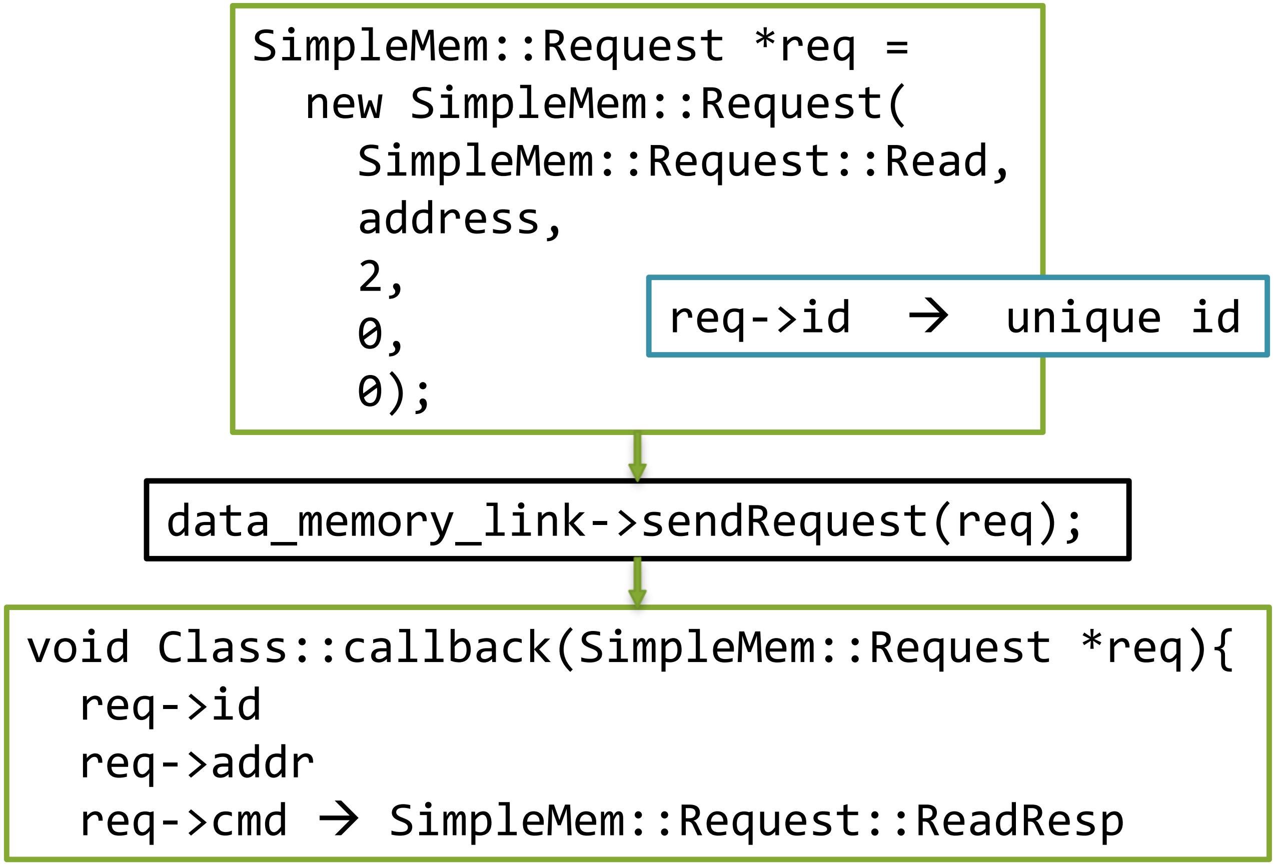
- **After the read/write request returns**
  - Read/write your own implementation of memory
    - E.g. an array std::array<uint8_t,65536> memory;

```
void function(addr) {
  // Read done
  memory[addr]=data & 0x00FF;
  memory[addr+1]=(data>>8) & 0x00FF;
}
```

CPU simulation

The memory interface

- **Read**



```
SimpleMem::Request *req =
  new SimpleMem::Request(
    SimpleMem::Request::Read,
    address,
    2,
    0,
    0);
```

```
req->id  →  unique id
```

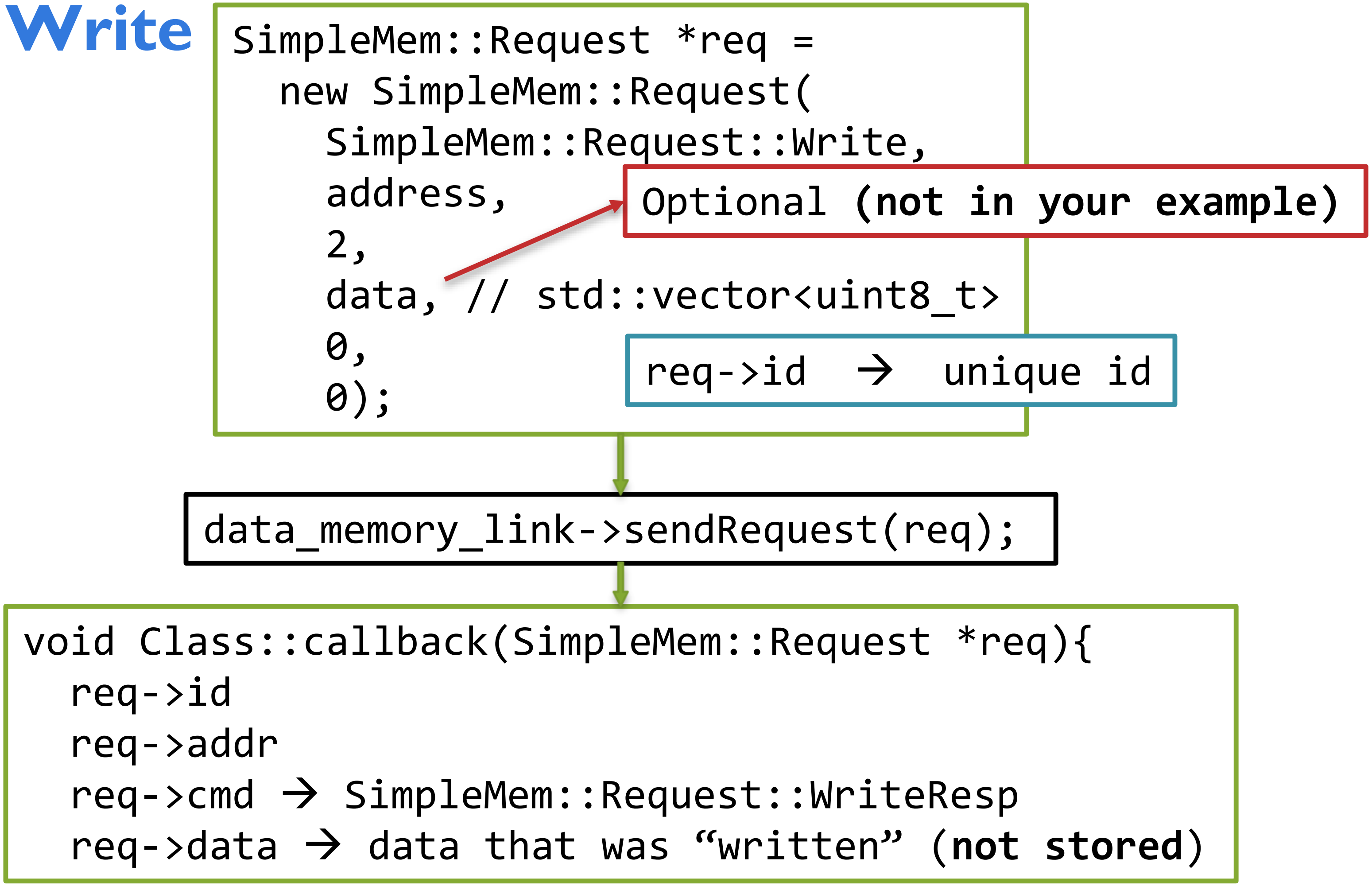
```
data_memory_link->sendRequest(req);
```

```
void Class::callback(SimpleMem::Request *req){
  req->id
  req->addr
  req->cmd → SimpleMem::Request::ReadResp
```

CPU simulation

The memory interface

- **Write**



```
SimpleMem::Request *req =
  new SimpleMem::Request(
    SimpleMem::Request::Write,
    address,
    2,
    data, // std::vector<uint8_t>
    0,
    0);
```

Optional **(not in your example)**

req->id → unique id

```
data_memory_link->sendRequest(req);
```

```
void Class::callback(SimpleMem::Request *req){
  req->id
  req->addr
  req->cmd → SimpleMem::Request::WriteResp
  req->data → data that was “written” (not stored)
```