Modeling memory objects in gem5: Ports

IMPORTANT: This slide deck builds on top of what has already been developed in Introduction to SimObjects, Debugging gem5, and Event Driven Simulation.

Ports

In gem5, `SimObjects` can use `Ports` to send/request data. `Ports` are gem5's **main interface to the memory**. There are two types of `Ports` in gem5: `RequestPort` and `ResponsePort`.

As their names would suggest:

- `RequestPorts` make `requests` and await `responses`.
- `ResponsePorts` await `requests` and send `responses`.

Make sure to differentiate between `request`/`response` and `data`. Both `requests` and `response` can carry `data` with them.

Packets

`Packets` facilitate communication through ports. They can be either `request` or `response` packets.

> **NOTE**: `Packet` in gem5 can change from a `request` to a `response`. This happens when the `request` arrives at a `SimObject` that can respond to it.

Every `Packet` has the following fields:

- `Addr`: Address of the memory location being accessed.
- `Data`: Data associated with the `Packet` (the data that `Packet` carries).
- `MemCmd`: Denotes the kind of `Packet` and what it should do.
  - Examples include: `readReq`/ `readResp`/ `writeReq`/ `writeResp`.
- `RequestorID`: ID for the `SimObject` that created the request (requestor).

Class `Packet` is defined in `gem5/src/mem/packet.hh`. Note that, in our tutorial, we will deal with `Packet` in pointers. `PacketPtr` is a type in gem5 that is equivalent to `Packet*`.

Ports in gem5

Let's take a look at `gem5/src/mem/port.hh` to see the declarations for `Port` classes.

Let's focus on the following functions. These functions make communication possible. Notice how `recvTimingReq` and `recvTimingResp` are `pure virtual` functions. This means that you can **not** instantiate an object of `RequestPort` or `ResponsePort` and instead you must extend them to fit your use case.

```
class RequestPort {
    bool sendTimingReq(PacketPtr pkt);
    // inherited from TimingRequestProtocol
    // in `src/mem/protocol/timing.hh`
    virtual bool recvTimingResp(PacketPtr pkt) = 0;
    virtual void sendRetryResp();
};
```

```
class ResponsePort {
    bool sendTimingResp(PacketPtr pkt);
    // inherited from TimingResponseProtocol in
    // `src/mem/protocol/timing.hh`
    virtual bool recvTimingReq(PacketPtr pkt) = 0;
    virtual void sendRetryReq();
};
```

Access Modes: Timing, Atomic, Functional

`Ports` allow 3 memory access modes:

1: In **`timing`** mode, accesses advance simulator time. In this mode, `requests` propagate down the memory hierarchy while each level imposes its latency and can potentially interleave processing of multiple requests. This mode is the only realistic mode in accessing the memory.
2: In **`atomic`** mode, accesses do not directly advance simulator time, rather it's left to the **original** `requestor` to move simulator time. Accesses are done atomically (are **not** interleaved). This access mode is useful for fast-forwarding simulation.
3: In **`functional`** mode, accesses to the memory are done through a chain of function calls. `Functional` mode does not advance simulator time. All accesses are done in series and are not interleaved. This access mode is useful for initializing simulation from files, i.e. talking from the host to the simulator.

Timing Protocol in Action

> **IMPORTANT**: A `Port` can only be connected to **one other** `Port` and it must be of a different type: `RequestPort`/`ResponsePort` can only be connected to `ResponsePort`/`RequestPort`.

If you look at `gem5/src/mem/port.hh` you'll see that class `RequestPort` has a `private` member called `ResponsePort* _responsePort` that holds a pointer to the `ResponsePort` that the `RequestPort` object is connected to (its `peer`).

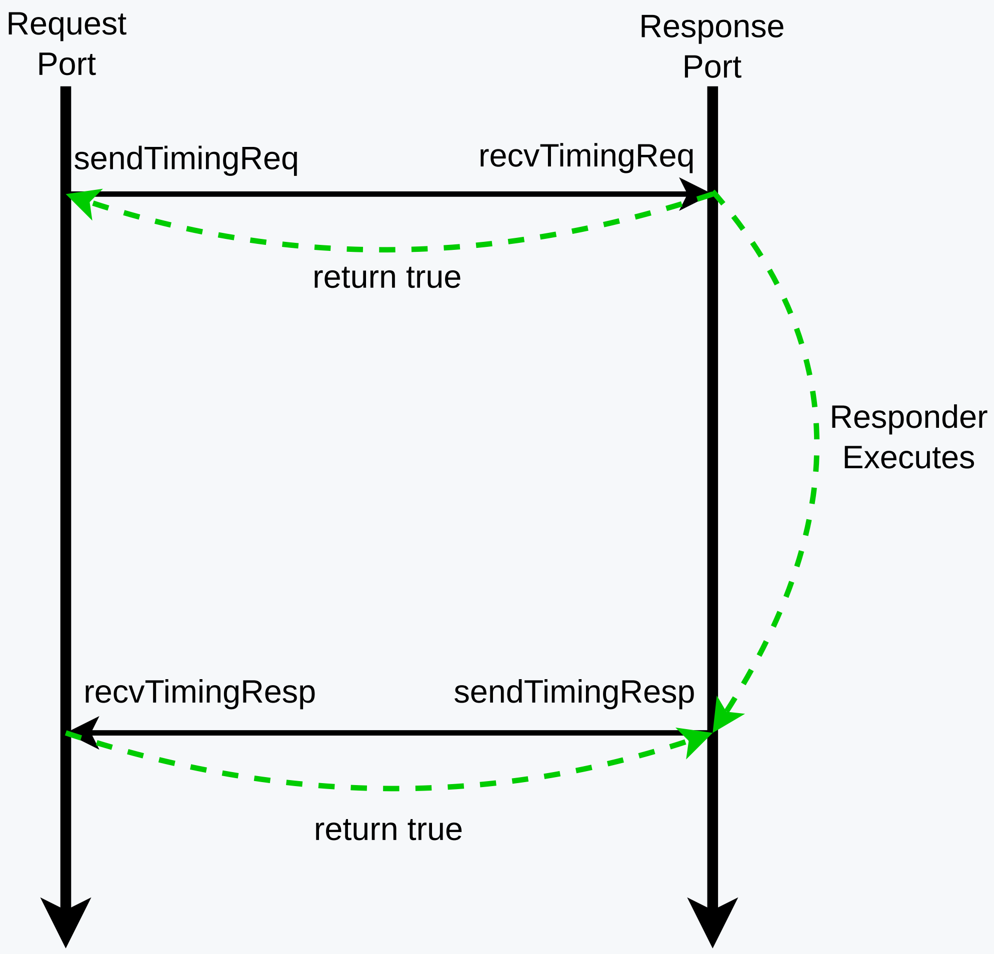
Moreover, if you look at the definition of `sendTimingReq`/`sendTimingResp` in `gem5/src/mem/port.hh` you'll see that they will call and return `peer::recvTimingReq`/`peer::recvTimingResp`.

Now let's look at 2 scenarios for communication, in these scenarios let's assume:

- `Requestor` is a `SimObject` that has a `RequestPort`.
- `Responder` is a `SimObject` that has a `ReponsePort`.

NOTE: Note that while in our scenarios `Requestor` and `Responder` have one `Port`, `SimObjects` can have multiple ports of different types.

Scenario: Everything Goes Smoothly



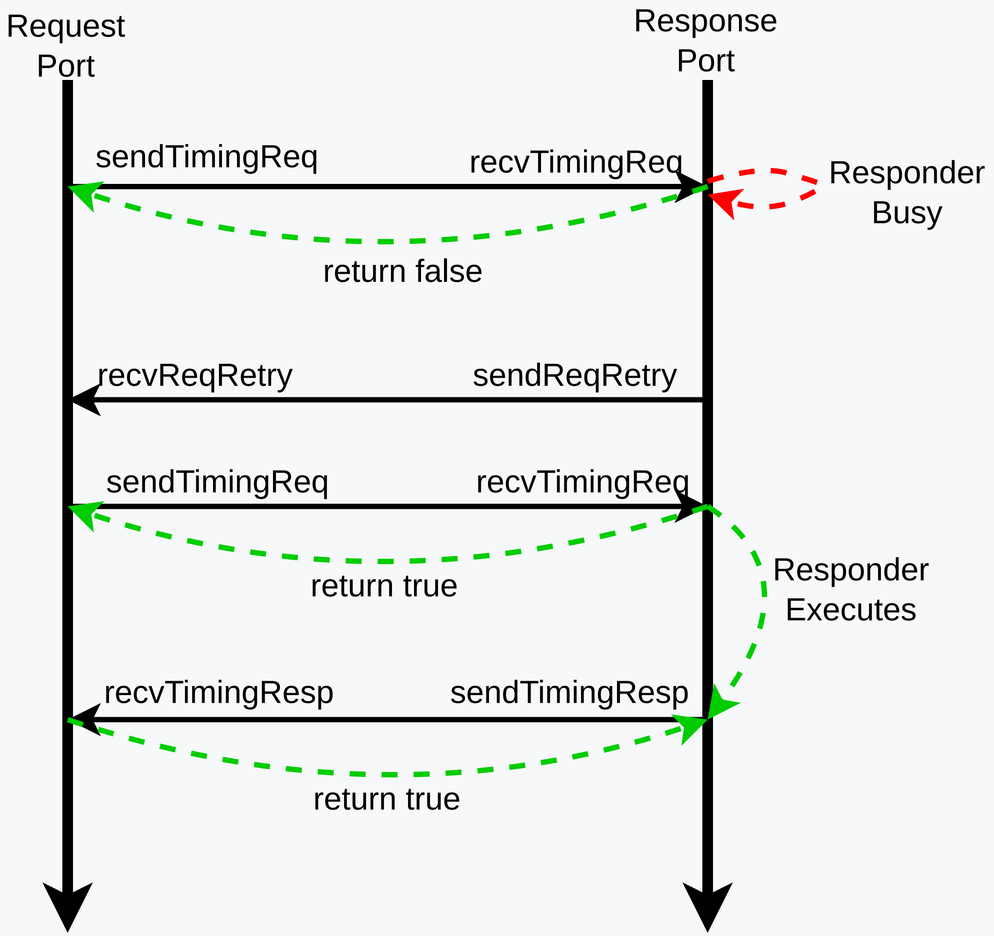
Scenario: Everything Goes Smoothly (details)

In this scenario:

1: **`Requestor`** sends a `Packet` as the `request` (e.g. a `readReq`). In C++ terms `Requestor::RequestPort::sendTimingReq` is called which in turn calls `Responder::ResponsePort::recvTimingReq`.
2: **`Responder`** is not busy and accepts the `request`. In C++ terms `Responder::ResponsePort::recvTimingReq` returns **true**. Since `Requestor` has received true, it will receive a `response` in the future.
3: Simulator time advances, `Requestor` and `Responder` continues execution. When `Responder` has the `response` (e.g. `readResp`) ready, it will send the `response` to the `requestor`. In C++ terms `Responder::ResponsePort::sendTimingResp` is called which in turn calls `Requestor::RequestPort::recvTimingResp`.
4: **`Requestor`** is not busy and accepts the `response`. In C++ terms `Requestor::RequestPort::recvTimingResp` returns true. Since `Responder` has received true, the transaction is complete.

gem5

Scenario: Responder Is Busy: Diagram



Scenario: Responder Is Busy

In this scenario:

1: **`Requestor`** sends a `Packet` as the `request` (e.g. a `readReq`).
2: **`Responder`** is busy and rejects the `request`. In C++ terms `Responder::ResponsePort::recvTimingReq` returns **false**. Since `Requestor` has received false, it waits for a `retry request` from `Responder`.
3: When **`Responder`** becomes available (is not busy anymore), it will send a `retry request` to `Requestor`. In C++ terms `Responder::ResponsePort::sendReqRetry` is called which in turn calls `Requestor::RequestPort::recvReqRetry`.
4: **`Requestor`** sends the `blocked Packet` as the `request` (e.g. a `readReq`).
5: **`Responder`** is not busy and accepts the `request`.
6: Simulator time advances, `Requestor` and `Responder` continue execution. When `Responder` has the `response` ready it will send the `response` to the `Requestor`.
7: **`Requestor`** is not busy and can accept the `response`.

Other Scenarios

There are two other possible scenarios:

1- A scenario where the `Requestor` is busy.
2- A scenario where both `Requestor` and `Responder` are busy.

CAUTION: Scenarios where `Requestor` is busy should not happen normally. In reality, the `Requestor` makes sure it can receive the `response` for a `request` when it sends the request. I have never run into a situation where I had to design my `SimObjects` in a way that the `Requestor` will return false when `recvTimingResp` is called. That's not to say that if you find yourself in a situation like this, you have done something wrong; BUT I would look really hard into my code/design and verify I'm simulating something realistic.

SecureMemory

In this step, we will implement our new `SimObject` called `SecureMemory`. `SecureMemory` will monitor all the traffic to the memory and make sure all the traffic is safe. In this tutorial, we will do this in multiple steps as laid out below.

Step 1:

We will implement `SecureMemory` to forward traffic from CPU to memory and back, causing latency for queueing traffic.

Step 2:

We will implement `SecureMemory` to check the Merkle tree of the data being sent to the memory.

Sidebar to Secure Memory

Step 1: Buffering Traffic

In this step, we will implement `SecureMemory` to forward traffic from CPU to memory and back, causing latency for queueing traffic.

We will do this in the following steps:

1. Create a `SimObject` declaration file for `SecureMemory`.
2. Add `Ports` to `SecureMemory`.
3. Add `FIFOs` to buffer traffic.
4. Implement the `getPort` function to map `Ports` in Python to `Ports` in C++.
5. Implement functions from `CPUSidePort` and `MemSidePort`.
6. Implement functions from `SecureMemory`.

ClockedObject

A `ClockedObject` is a child class of `SimObject` that provides facilities for managing time in `cycles`. Every `ClockedObject` has a `clk_domain` parameter that defines its clock frequency. Using the `clk_domain`, the `ClockedObject` provides functionalities like below:

- `clockEdge(Cycles n)`: A function that returns the time of the `nth` clock edge into the future.
- `nextCycle()`: A function that returns the time of first clock edge into the future, i.e. `nextCycle() := clockEdge(Cycles(1))`.

This class is defined in `gem5/src/sim/clocked_object.hh` as shown below:

```
class ClockedObject : public SimObject, public Clocked
{
  public:
    ClockedObject(const ClockedObjectParams &p);

    /** Parameters of ClockedObject */
    using Params = ClockedObjectParams;

    void serialize(CheckpointOut &cp) const override;
    void unserialize(CheckpointIn &cp) override;

    PowerState *powerState;
};
```

SecureMemory: Adding Files

Now let's go ahead and create a `SimObject` declaration file for `SecureMemory`. Do it by running the following commands:

```
cd /workspaces/2024/gem5/src
mkdir -p bootcamp/inspector-gadget
cd bootcamp/inspector-gadget
touch SecureMemory.py
```

Now, let's also create a `SConscript` for registering `SecureMemory`. Do it by running the following command:

```
touch SConscript
```

InspectoGadget: SimObject Declaration File

Now, inside `SecureMemory.py`, let's define `SecureMemory` as a `ClockedObject`. To do that, we need to import `ClockedObject`. Do it by adding the following line to `SecureMemory.py`.

```
from m5.objects.ClockedObject import ClockedObject
```

The remaining part of the declaration is for now similar to that of `HelloSimObject` in Introduction to SimObjects. Do that part on your own. When you are done, you can find my version of the code in the next slide.

SecureMemory: SimObject Declaration File So Far

This is what should be in `SecureMemory.py` now:

```python
from m5.objects.ClockedObject import ClockedObject

class SecureMemory(ClockedObject):
    type = "SecureMemory"
    cxx_header = "bootcamp/secure-memory/inspector_gadget.hh"
    cxx_class = "gem5::SecureMemory"
```

SecureMemory: Ports in Python

So far we have looked at the declaration of `Ports` in C++. However, to create an instance of a C++ class in Python we need a declaration of that class in Python. `Ports` are defined under `gem5/src/python/m5/params.py`. However, `Ports` do not inherit from class `Param`. I strongly recommend that you take a short look at `gem5/src/python/m5/params.py`.

Try to find what kind of parameters you can add to any `SimObject`/`ClockedObject`.

Our next step is to define a `RequestPort` and a `ResponsePort` for `SecureMemory`. To do this add the following import line to `SecureMemory.py`.

```python
from m5.params import *
```

NOTE: My personal preference in Python is to import modules very explicitly. However, when importing `m5.params`, I think it's ok to do import `*`. This is mainly because, when I'm creating `SimObjects`, I might need different kinds of parameters that I might not know about in advance.

SecureMemory: Adding Ports

Now, let's finally add two ports to `SecureMemory`; One port will be on the side where the CPU would be in the computer system and one port will be on the side where the memory would be. Therefore, let's call them `cpu_side_port` and `mem_side_port` respectively.

Question: What type should `cpu_side_port` and `mem_side_port` be?

Before looking at the answer, try to answer the question for yourself.

Answer: `cpu_side_port` should be a `ResponsePort` and `mem_side_port` should be a `RequestPort`.

Make sure this answer makes sense to you before moving on to the next slide.

SecureMemory: Adding Ports cont.

Add the following two lines under the declaration of `SecureMemory` to add `cpu_side_port` and `mem_side_port`:

```
cpu_side_port = ResponsePort("ResponsePort to receive requests from CPU side.")
mem_side_port = RequestPort("RequestPort to send received requests to memory side.")
```

To buffer traffic, we need two FIFOs: one for `requests` (from `cpu_side_port` to `mem_side_port`) and one for `responses` (from `mem_side_port` to `cpu_side_port`). For the the FIFO in the `request` path, we know that in the future we want to *inspect* the requests. Therefore, let's call it `buffer`. We also need a parameter to determine the the number of entries in this buffer so let's call that parameter `inspection_buffer_entries`. For the `response` path, we will simply call the buffer `response_buffer` and add a parameter for its entries named `response_buffer_entries`. Do it by adding the following lines under the declaration of `SecureMemory`:

```
inspection_buffer_entries = Param.Int("Number of entries in the inspection buffer.")
response_buffer_entries = Param.Int("Number of entries in the response buffer.")
```

SecureMemory: SimObject Declaration File

This is what should be in `SecureMemory.py` now:

```
from m5.objects.ClockedObject import ClockedObject
from m5.params import *


class SecureMemory(ClockedObject):
    type = "SecureMemory"
    cxx_header = "bootcamp/secure-memory/inspector_gadget.hh"
    cxx_class = "gem5::SecureMemory"

    cpu_side_port = ResponsePort("ResponsePort to received requests from CPU side.")
    mem_side_port = RequestPort("RequestPort to send received requests to memory side.")

    inspection_buffer_entries = Param.Int("Number of entries in the inspection buffer.")
    response_buffer_entries = Param.Int("Number of entries in the response buffer.")
```

Updating SConscript

Remember to register `SecureMemory` as a `SimObject` as well as create a `DebugFlag` for it.

To do this, put the following in `gem5/src/bootcamp/secure-memory/SConscript`:

```
Import("*")

SimObject("SecureMemory.py", sim_objects=["SecureMemory"])

Source("inspector_gadget.cc")

DebugFlag("SecureMemory")
```

> **NOTE**: In the next steps we will create `inspector_gadget.hh` and `inspector_gadget.cc`.

SecureMemory: C++ Files

Now, let's go ahead and create a header and source file for `SecureMemory` in `gem5/src/bootcamp/inspector-gadget`. Remember to make sure the path to your header file matches that of what you specified in `cxx_header` in `SecureMemory.py` and the path for your source file matches that of what you specified in `SConscript`. Run the following commands from within our `inspector-gadget` directory:

```
touch inspector_gadget.hh
touch inspector_gadget.cc
```

Now, let's simply declare `SecureMemory` as a class that inherits from `ClockedObject`. This means you have to import `sim/clocked_object.hh` instead of `sim/sim_object.hh`. Let's add everything that we have added in the Python to our class except for the `Ports`.

SecureMemory: Header File

```
#ifndef __BOOTCAMP_INSPECTOR_GADGET_INSPECTOR_GADGET_HH__
#define __BOOTCAMP_INSPECTOR_GADGET_INSPECTOR_GADGET_HH__

#include "params/SecureMemory.hh"
#include "sim/clocked_object.hh"

namespace gem5
{

class SecureMemory : public ClockedObject
{
  private:
    int bufferEntries;
    int responseBufferEntries;

  public:
    SecureMemory(const SecureMemoryParams& params);
};


} // namespace gem5

#endif // __BOOTCAMP_INSPECTOR_GADGET_INSPECTOR_GADGET_HH__
```

Extending Ports

Recall, `RequestPort` and `ResponsePort` classes were abstract classes, i.e. they had `pure virtual` functions which means objects cannot be instantiated from that class. Therefore, for us to use `Ports` we need to extend the classes and implement their `pure virtual` functions.

Before anything, let's go ahead and import the header file that contains the declaration for `Port` classes. We also need to include `mem/packet.hh` since we will be dealing with and moving around `Packets` a lot. Do it by adding the following lines to `inspector_gadget.hh`:

```
#include "mem/packet.hh"
#include "mem/port.hh"
```

> **REMEMBER** to follow the right include order based on gem5's convention.

Extending ResponsePort

Now, let's get to extending `ResponsePort` class. Let's do it inside the scope of our `SecureMemory` class to prevent using names used by other gem5 developers. Let's go ahead an create `CPUSidePort` class that inherits from `ResponsePort` in the `private` scope. To do this, add the following code to `inspector_gadget.hh`.

```
private:
  class CPUSidePort: public ResponsePort
  {
    private:
      SecureMemory* owner;
      bool needToSendRetry;
      PacketPtr blockedPacket;

    public:
      CPUSidePort(SecureMemory* owner, const std::string& name):
          ResponsePort(name), owner(owner), needToSendRetry(false), blockedPacket(nullptr)
      {}
      bool needRetry() const { return needToSendRetry; }
      bool blocked() const { return blockedPacket != nullptr; }
      void sendPacket(PacketPtr pkt);

      virtual AddrRangeList getAddrRanges() const override;
      virtual bool recvTimingReq(PacketPtr pkt) override;
      virtual Tick recvAtomic(PacketPtr pkt) override;
      virtual void recvFunctional(PacketPtr pkt) override;
      virtual void recvRespRetry() override;
  };
```

Extending ResponsePort: Deeper Look

Here is a deeper look into the declaration of `CPUSidePort`.

1: `SecureMemory* owner` is a pointer to the instance of `SecureMemory` that owns this instance of `CPUSidePort`. We need to access the owner when we receive `requests`, i.e. when `recvTimingReq` is called.
2: `bool needToSendRetry` tells us if we need to send a `retry request`. This happens when we reject a `request` because we are busy. When we are not busy, we check this before sending a `retry request`.
3: In addition to all the functions that are used for moving packets, the class `ResponsePort` has another `pure virtual` function that will return an `AddrRangeList` which represents all the address ranges for which the port can respond. Note that, in a system, the memory addresses can be partitioned among ports. Class `RequestPort` has a function with the same name, but it is already implemented and will just ask its peer `ResponsePort` for the ranges by calling `peer::getAddrRanges`.
4: We will need to implement all of the functions that relate to moving packets – the ones that start with `recv`. We will use `owner` to implement most of the functionality of these functions within `SecureMemory`.
5: We'll talk about `PacketPtr blockedPacket` in the next slides.

Extending RequestPort

We're going to follow a similar approach for extending `RequestPort`. Let's create class `MemSidePort` that inherits from `RequestPort`. Again we'll do it in the `private` scope of `SecureMemory`. Do it by adding the following code to `inspector_gadget.hh`.

```
private:
  class MemSidePort: public RequestPort
  {
    private:
      SecureMemory* owner;
      bool needToSendRetry;
      PacketPtr blockedPacket;

    public:
      MemSidePort(SecureMemory* owner, const std::string& name):
          RequestPort(name), owner(owner), needToSendRetry(false), blockedPacket(nullptr)
      {}
      bool needRetry() const { return needToSendRetry; }
      bool blocked() const { return blockedPacket != nullptr; }
      void sendPacket(PacketPtr pkt);

      virtual bool recvTimingResp(PacketPtr pkt) override;
      virtual void recvReqRetry() override;
  };
```

Extending RequestPort: Deeper Look

Let's take a deeper look into what we added for class `MemSidePort`.

1: Like `CPUSidePort`, a `MemSidePort` instance holds a pointer to its `owner` with `SecureMemory* owner`. We do this to access the owner when we receive `responses`, i.e. when `recvTimingResp` is called.
2: When `MemSidePort::sendTimingReq` receives false, it means the request was blocked. We track a pointer to this blocked `Packet` in `PacketPtr blockedPacket` so that we can retry the request later.
3: Function `blocked` tells us if we are blocked by the memory side, i.e. still waiting to receive a `retry request` from memory side.
4: Function `sendPacket` is a wrapper around `sendTimingReq` to give our code more structure. Notice we don't need to definte `sendTimingReq` as it is already defined by `TimingRequestProtocol`.
5: We will need to implement all of the functions that relate to moving packets – the ones that start with `recv`. We will use `owner` to implement most of the functionality of these functions within `SecureMemory`.

Creating Instances of Ports in SecureMemory

Now that we have declared `CPUSidePort` and `MemSidePort` classes (which are not abstract classes), we can go ahead and create an instance of each class in `SecureMemory`. To do that, add the following two lines to `inspector_gadget.hh`

```
private:
  CPUSidePort cpuSidePort;
  MemSidePort memSidePort;
```

SimObject::getPort

Let's take a look at `gem5/src/sim/sim_object.hh` again. Below is a snippet of code from the declaration of the function `getPort` in the class `SimObject`.

```
  public:
/**
     * Get a port with a given name and index. This is used at binding time
     * and returns a reference to a protocol-agnostic port.
     *
     * gem5 has a request and response port interface. All memory objects
     * are connected together via ports. These ports provide a rigid
     * interface between these memory objects. These ports implement
     * three different memory system modes: timing, atomic, and
     * functional. The most important mode is the timing mode and here
     * timing mode is used for conducting cycle-level timing
     * experiments. The other modes are only used in special
     * circumstances and should *not* be used to conduct cycle-level
     * timing experiments. The other modes are only used in special
     * circumstances. These ports allow SimObjects to communicate with
     * each other.
     *
     * @param if_name Port name
     * @param idx Index in the case of a VectorPort
     *
     * @return A reference to the given port
     *
     * @ingroup api_simobject
     */
    virtual Port &getPort(const std::string &if_name, PortID idx=InvalidPortID);
```

SimObject::getPort cont.

This function is used for connecting ports to each other. As far as we are concerned, we need to create a mapping between our `Port` objects in C++ and the `Ports` that we declare in Python. To the best of my knowledge, we will never have to call this function on our own.

For now, let's implement this function to return a `Port&` when we recognize `if_name` (which would be the name that we gave to a `Port` in Python) and, otherwise, we will pass this up to the parent class `ClockedObject` to handle the function call.

Let's go ahead and add the declaration for this function to `inspector_gadget.hh`.

```
  public:
    virtual Port& getPort(const std::string& if_name, PortID idxInvalidPortID);
```

Enough with the Declarations! For Now!

So far, we have declared quite a few functions that we need to implement. Let's start defining some of them. In the next several slides, we will be defining functions from `CPUSidePort` and `MemSidePort`, as well as `getPort` from `SecureMemory`.

Open `inspector_gadget.cc` and let's start by adding the following include statements:

```
#include "bootcamp/secure-memory/inspector_gadget.hh"

#include "debug/SecureMemory.hh"
```

As we start defining functions, we will likely find the need to declare and define more functions. To keep things organized, let's just note them down as we go. We will then go back to declaring and defining them.

Defining SecureMemory::getPort

Let's start by implementing `SecureMemory::getPort`. Add the following code inside `namespace gem5` in `inspector_gadget.cc` to do this (all code in such definition files should go inside `namespace gem5`):

```
Port&
SecureMemory::getPort(const std::string &if_name, PortID idx)
{
    if (if_name == "cpu_side_port") {
        return cpuSidePort;
    } else if (if_name == "mem_side_port") {
        return memSidePort;
    } else {
        return ClockedObject::getPort(if_name, idx);
    }
}
```

If you remember, `getPort` needs to create a mapping between `Port` objects in Python and `Port` objects in C++. In this function when `if_name == "cpu_side_port"` we will return `cpuSidePort` (the name comes from the Python declaration, look at `SecureMemory.py`) . We do the same thing to map `"mem_side_port"` to our instance `memSidePort`. For now, you don't have to worry about `idx`. We will talk about it later in the context of `VectorPorts` – ports that can connect to multiple peers.

Defining Functions from CPUSidePort

Now, that we have implemented `SecureMemory::getPort`, we can start declaring and defining the functions that simulate the `request` path (from `cpu_side_port` to `mem_side_port`) in `SecureMemory`. Here are all the functions that we need to define from `CPUSidePort`:

```
virtual AddrRangeList getAddrRanges() const override;

virtual bool recvTimingReq(PacketPtr pkt) override;
virtual Tick recvAtomic(PacketPtr pkt) override;
virtual void recvFunctional(PacketPtr pkt) override;
```

As we start defining these functions you will see that `Ports` are interfaces that facilitate communication between `SimObjects`. Most of these functions rely on `SecureMemory` to provide the bulk of the functionality.

CPUSidePort::recvAtomic, CPUSidePort::recvFunctional

These two functions are very simple to define. Basically, our responsibility is to pass the `PacketPtr` to `SimObjects` further down in the memory hierarchy. To implement them we will call functions with the same name from `SecureMemory`. Add the following code to `inspector_gadget.cc`:

```
Tick
SecureMemory::CPUSidePort::recvAtomic(PacketPtr pkt)
{
    DPRINTF(SecureMemory, "%s: Received pkt: %s in atomic mode.\n", __func__, pkt->print());
    return owner->recvAtomic(pkt);
}

void
SecureMemory::CPUSidePort::recvFunctional(PacketPtr pkt)
{
    DPRINTF(SecureMemory, "%s: Received pkt: %s in functional mode.\n", __func__, pkt->print());
    owner->recvFunctional(pkt);
}
```

We will also need to eventually declare the functions that we call from the owner, which is an `SecureMemory`. Keep these in mind:

Declarations:

`Tick SecureMemory::recvAtomic(PacketPtr);`

`void SecureMemory::recvFunctional(PakcetPtr);`

CPUSidePort::getAddrRanges

Reminder: This function returns an `AddrRangeList` that represents the address ranges for which the port is a responder. To understand this better, think about dual channel memory. Each channel in the memory is responsible for a subset of all the addresses in your computer.

To define this function, we are again going to rely on `SecureMemory` and call a function with the same name from `SecureMemory`. Do this by adding the following code to `inspector_gadget.cc`:

```
AddrRangeList
SecureMemory::CPUSidePort::getAddrRanges() const
{
    return owner->getAddrRanges();
}
```

Declarations:

`AddrRangeList SecureMemory::getAddrRanges() const;`

CPUSidePort::recvTimingReq

In this function we will do the following:

Ask the owner to receive the `Packet` the `Port` is receiving. To do this we will call a function with the same name from `SecureMemory`. If `SecureMemory` can accept the `Packet` then the `Port` will return true. Otherwise, the `Port` will return false as well as remember that we need to send a `retry request` in the future, i.e. we will set `needToSendRetry = true`.

To define this function add the following code to `inspector_gadget.cc`.

```
bool
SecureMemory::CPUSidePort::recvTimingReq(PacketPtr pkt)
{
    DPRINTF(SecureMemory, "%s: Received pkt: %s in timing mode.\n", __func__, pkt->print());
    if (owner->recvTimingReq(pkt)) {
        return true;
    }
    needToSendRetry = true;
    return false;
}
```

Declarations:

`bool SecureMemory::recvTimingReq(PacketPtr);`

Back to Declaration

Now that we are finished with defining functions from `CPUSidePort`, let's go ahead and declare the functions from `SecureMemory` that we noted down.

To do this add the following code to the `public` scope of `SecureMemory` in `inspector_gadget.hh`.

```
public:
  AddrRangeList getAddrRanges() const;
  bool recvTimingReq(PacketPtr pkt);
  Tick recvAtomic(PacketPtr pkt);
  void recvFunctional(PacketPtr pkt);
```

TimedQueue

As we mentioned, in this first step, all `SecureMemory` does is buffer the traffic, forwarding `requests` and `responses`. To do that, let's create a "first in first out" (FIFO) structure for `buffer` and `responseBuffer`. The purpose of this structure is impose a minimum latency between the times when items are pushed to the queue and when they can be accessed. We will add a member variable called `latency` to make this delay configurable. We will wrap `std::queue` to expose the following functionalities.

1: Method `front` that will return a reference to the oldest item in the queue, similar to `std::queue`.
2: Method `pop` that will remove the oldest item in the queue, similar to `std::queue`.
3: Method `push` that will add a new item to the queue as well as tracking the simulation time the item was inserted. This is useful for ensuring a minimum amount of time has passed before making it ready to be accessed, modeling the latency.
4: Method `empty` that will return true if queue is empty, similar to `std::queue`.
5: Method `size` that will return the number of items in the queue, similar to `std::queue`.
6: Method `hasReady` will return true if an item in the queue can be accessed at a given time (i.e. has spent a minimum latency in the queue).
7: Method `firstReadyTime` will return the time at which the oldest item becomes accessible.

Timed Queue: Details

Like `CPUSidePort` and `MemSidePort`, let's declare our class `TimedQueue` in the `private` scope of `SecureMemory`. Do this by adding the lines on the right side of this slide to `inspector_gadget.hh`.

Make sure to add the following include statement to the top of the file as well.

```
#include <queue>
```

```
private:
  template<typename T>
  class TimedQueue
  {
    private:
      Tick latency;

      std::queue<T> items;
      std::queue<Tick> insertionTimes;

    public:
      TimedQueue(Tick latency): latency(latency) {}

      void push(T item, Tick insertion_time) {
          items.push(item);
          insertionTimes.push(insertion_time);
      }
      void pop() {
          items.pop();
          insertionTimes.pop();
      }

      T& front() { return items.front(); }
      bool empty() const { return items.empty(); }
      size_t size() const { return items.size(); }
      bool hasReady(Tick current_time) const {
          if (empty()) {
              return false;
          }
          return (current_time - insertionTimes.front()) >= latency;
      }
      Tick firstReadyTime() { return insertionTimes.front() + latency; }
  };
```

buffer

Now, let's declare an instance of `TimedQueue` to buffer `PacketPtrs` that `SecureMemory` receives from `SecureMemory::cpuSidePort::recvTimingReq`. Add the following lines to the `private` scope of class `SecureMemory` to do this:

```
private:
  TimedQueue<PacketPtr> buffer;
```

Now that we have declared `buffer`, we are ready to define the following functions (these are already declared):

```
AddrRangeList getAddrRanges() const;
bool recvTimingReq(PacketPtr pkt);
Tick recvAtomic(PacketPtr pkt);
void recvFunctional(PacketPtr pkt);
```

> **NOTE**: For now we are focusing on the `request` path, i.e. we're not going to define `recvRespRetry` just yet.

Let's Get the Easy Ones Out the Way

Between the four functions, `getAddrRanges` and `recvFunctional` are the most straightforward to define. We just need to call the same functions from `memSidePort`. To define these two functions, add the following code under `namespace gem5` in `inspector_gadget.cc`:

```cpp
AddrRangeList
SecureMemory::getAddrRanges() const
{
    return memSidePort.getAddrRanges();
}

void
SecureMemory::recvFunctional(PacketPtr pkt)
{
    memSidePort.sendFunctional(pkt);
}
```

> **NOTE**: These two functions are already defined by `RequestPort` and we don't need to redefine them. Notice how, for `Ports`, you only have to define functions that relate to receiving signals.

SecureMemory::recvAtomic

Looking at `recvAtomic`, this function returns a value of type `Tick`. This value is supposed to represent the latency of the access if that access was done in singularity, i.e atomically/without being interleaved. **CAUTION**: This latency is not an accurate representation of the actual latency of the access in a real setup. In a real setup there are many accesses happening at the same time and most of the time accesses do not happen atomically.

Let's add *one* cycle to the latency of accesses from the lower level of memory hierarchy. To do this we are going to call `clockPeriod` from the parent class of `SecureMemory`, which is `ClockedObject`. This function returns the period of the `clk_domain` in `Ticks`. Add the following code to define of `SecureMemory::recvAtomic` in `inspector_gadget.cc`.

```
Tick
SecureMemory::recvAtomic(PacketPtr pkt)
{
    return clockPeriod() + memSidePort.sendAtomic(pkt);
}
```

On to the Hard Part

As we discussed before, `timing` accesses are the accesses that advance simulator time and represent real setups.

`SecureMemory::recvTimingReq` will need to check if there is at least one available entry in the `buffer`. If there are no entries left, it should return false; otherwise, it should place the `Packet` at the end of the buffer – i.e. `push` into `buffer` – and return true.

To define `SecureMemory::recvTimingReq`, add the following code to `secure-memory.cc`:

```
bool
SecureMemory::recvTimingReq(PacketPtr pkt)
{
    if (buffer.size() >= bufferEntries) {
        return false;
    }
    buffer.push(pkt, curTick());
    return true;
}
```

We're Not Done Yet!

So far, we have managed to program the movement of `Packets` from `cpuSidePort` into `buffer`. Now, we need to send the `Packets` that are inside `buffer` to `memSidePort`.

One would ask, why not call `memSidePort.sendTimingReq` inside `SecureMemory::recvTimingReq`? The answer is because we want to impose a latency on the movement of the `Packet` through `buffer`. Think about how the real hardware would work. If the `Packet` is available on `cpuSidePort` on the rising edge of the clock, it would go inside `buffer` by the falling edge of the clock, i.e. time will pass. Now, assuming that `Packet` is at the front of `buffer`, it will be available on the rising edge of the next clock cycle. If you remember, we use `events` to make things happen in the future by defining callback functions.

Now, let's go ahead and declare an `EventFunctionWrapper` for picking the `Packet` at the front of `buffer` and sending it through `memSidePort`.

nextReqSendEvent

We're going to declare a function `EventFunctionWrapper nextReqSendEvent` to send `Packets` through `memSidePort`. Remember what we need to do?

Add the following include statement to `inspector_gadget.hh` to include the appropriate header file for the class `EventFunctionWrapper`.

```
#include "sim/eventq.hh"
```

If you remember from Event Driven Simulation, we also need to declare a `std::function<void>()` to pass as the callback function for `nextReqSendEvent`. I would like to name these functions with `process` prefixing the name of the `event`. Let's go ahead and declare `nextReqSendEvent` as well as its callback function in the `private` scope of `SecureMemory`. Do it by adding the following lines to `inspector_gadget.hh`:

```
  private:
    EventFunctionWrapper nextReqSendEvent;
    void processNextReqSendEvent();
```

Managing the Schedule of nextReqSendEvent

Now, that we have declared `nextReqSendEvent`, we can schedule `nextReqSendEvent` in `SecureMemory::recvTimingReq`. We will see in a few slides why it is helpful to have a function that decides if and when `nextReqSendEvent` should be scheduled.

What I do when I write `SimObjects` is that, for every `event`, I create a function to schedule that event. I name these functions with `schedule` prefixing the name of the event. Let's go ahead and a declare `scheduleNextReqSendEvent` under the `private` scope in `SecureMemory`.

Open `inspector_gadget.hh` and add the following lines:

```
  private:
    void scheduleNextReqSendEvent(Tick when);
```

We'll see that one `event` might be scheduled in multiple locations in the code. At every location, we might have a different perspective on when an `event` should be scheduled. `Tick when` denotes the earliest we think that `event` should be scheduled from the perspective of the location.

Back to SecureMemory::recvTimingReq

Now, we can finally go ahead and add a function call to `scheduleNextReqSendEvent` in `SecureMemory::recvTimingReq`. Since we are assuming it will take **one** `cycle` to insert an item to `buffer`, we're going to pass `nextCycle()` as `when` argument.

This is how `SecureMemory::recvTimingReq` should look after all the changes.

```
bool
SecureMemory::recvTimingReq(PacketPtr pkt)
{
    if (buffer.size() >= bufferEntries) {
        return false;
    }
    buffer.push(pkt, curTick());
    scheduleNextReqSendEvent(nextCycle());
    return true;
}
```

MemSidePort::sendPacket

As mentioned before, it's a good idea to create a function for sending `Packets` through `memSidePort`. To do this, let's go ahead and define `MemSidePort::sendPacket`. We define this function now since we're going to need it in `processNextReqSendEvent`.

To define `MemSidePort::sendPacket` add the following code to `inspector_gadget.cc`

```
void
SecureMemory::MemSidePort::sendPacket(PacketPtr pkt)
{
    panic_if(blocked(), "Should never try to send if blocked!");

    DPRINTF(SecureMemory, "%s: Sending pkt: %s.\n", __func__, pkt->print());
    if (!sendTimingReq(pkt)) {
        DPRINTF(SecureMemory, "%s: Failed to send pkt: %s.\n", __func__, pkt->print());
        blockedPacket = pkt;
    }
}
```

NOTE: We call `panic` if this function is called when we have a blocked `Packet`. This is because if there is already a `Packet` that is rejected, we expect consequent `Packets` be rejected until we receive a `retry request`. We make sure to follow this by not trying to send `Packets` when blocked prior.

SecureMemory::processNextReqSendEvent cont.

Now that we have defined `sendPacket`, we can use it in `processNextReqSendEvent`. Add the following definition to `inspector_gadget.cc`:

```
void
SecureMemory::processNextReqSendEvent()
{
    panic_if(memSidePort.blocked(), "Should never try to send if blocked!");
    panic_if(!buffer.hasReady(curTick()), "Should never try to send if no ready packets!");

    PacketPtr pkt = buffer.front();
    memSidePort.sendPacket(pkt);
    buffer.pop();

    scheduleNextReqSendEvent(nextCycle());
}
```

SecureMemory::processNextReqSendEvent: Deeper Look

Here are a few things to note about `processNextReqSendEvent`:

1: We should not try to send a `Packet` if `memSidePort.blocked()` is true. We made this design decision and checked for it in `MemSidePort::sendPacket` to prevent `Packets` from being lost or accidentally changing the order of `Packets`.
2: We should not try to send a `Packet` when there is no `Packet` ready at `curTick()`.
3: When we are done, we need to try to schedule `nextReqSendEvent` in its callback event.

> Let's take a step back...

Are we done with `cpuSidePort` yet? If we look at `SecureMemory::recvTimingReq`, we return false when there is not enough space in `buffer`. Also, if you remember, if the `reponsder` (in our case `SecureMemory`) rejects a `request` because it's busy (in our case because we don't have enough space in `buffer`), the `responder` has to send a `request retry` when it becomes available (in our case, when there is room freed in `buffer`). So let's go ahead and send a `request retry` to the `peer` of `cpuSidePort`. We need to send that retry **one cycle later**. So, we need another event for that. Let's go ahead and add it.

nextReqRetryEvent

Let's add a declaration for `nextReqRetryEvent` as well as its callback function and its scheduler function. To do it add the following lines to the `private` scope of `SecureMemory` in `inspector_gadget.hh`.

```
  private:
    EventFunctionWrapper nextReqRetryEvent;
    void processNextReqRetryEvent();
    void scheduleNextReqRetryEvent(Tick when);
```

Define the functions by adding the following code in `inspector_gadget.cc`.

```
void
SecureMemory::processNextReqRetryEvent()
{
    panic_if(!cpuSidePort.needRetry(), "Should never try to send retry if not needed!");
    cpuSidePort.sendRetryReq();
}

void
SecureMemory::scheduleNextReqRetryEvent(Tick when)
{
    if (cpuSidePort.needRetry() && !nextReqRetryEvent.scheduled()) {
        schedule(nextReqRetryEvent, align(when));
    }
}
```

Back to processNextReqSendEvent

Now all that is left to do in `processNextReqSendEvent` is to try scheduling `nextReqRetry` for `nextCycle` after we have sent a `Packet`. Let's go ahead and add that our code. This is how `processNextReqSendEvent` should look like after these changes:

```
void
SecureMemory::processNextReqSendEvent()
{
    panic_if(memSidePort.blocked(), "Should never try to send if blocked!");
    panic_if(!buffer.hasReady(curTick()), "Should never try to send if no ready packets!");

    PacketPtr pkt = buffer.front();
    memSidePort.sendPacket(pkt);
    buffer.pop();

    scheduleNextReqRetryEvent(nextCycle());
    scheduleNextReqSendEvent(nextCycle());
}
```

Next we will see the details of the scheduler function for `nextReqSendEvent`.

scheduleNextReqSendEvent

To define `scheduleNextReqSendEvent`, add the following code to `inspector_gadget.cc`.

```
void
SecureMemory::scheduleNextReqSendEvent(Tick when)
{
    bool port_avail = !memSidePort.blocked();
    bool have_items = !buffer.empty();

    if (port_avail && have_items && !nextReqSendEvent.scheduled()) {
        Tick schedule_time = align(buffer.firstReadyTime());
        schedule(nextReqSendEvent, schedule_time);
    }
}
```

You might wonder why we need to calculate `schedule_time` ourselves. As we mentioned, `Tick when` is passed from the perspective of the caller for when it thinks `nextReqSendEvent` should be scheduled. However, we need to make sure that we schedule the event at the time that simulates latencies correctly.

Make sure to add the following include statement as well since we're using `std::max`.

```
#include <algorithm>
```

MemSidePort::recvReqRetry

We're almost done with defining the whole `request` path. The only thing that remains is to react to `request retries` we receive from the `peer` of `memSidePort`.

Since we tracked the last `Packet` that we have tried to send, we can simply try sending that packet again. Let's consider the following for this function:

1: We shouldn't receive a `request retry` if we're not blocked.
2: For now, let's accept that there might be scenarios when a `request retry` will arrive but when we try to send `blockedPacket`, it will be rejected again. So let's account for that when writing `MemSidePort::recvReqRetry`.
3: If sending `blockedPacket` succeeds, we can now try to schedule `nextReqSendEvent` for `nextCycle` (we have to ask `owner` to do this).

MemSidePort::recvReqRetry cont.

Add the following code to `inspector_gadget.cc` to define `MemSidePort::recvReqRetry`

```
void
SecureMemory::MemSidePort::recvReqRetry()
{
    panic_if(!blocked(), "Should never receive retry if not blocked!");

    DPRINTF(SecureMemory, "%s: Received retry signal.\n", __func__);
    PacketPtr pkt = blockedPacket;
    blockedPacket = nullptr;
    sendPacket(pkt);

    if (!blocked()) {
        owner->recvReqRetry();
    }
}
```

Declarations:

`void SecureMemory::recvReqRetry();`

SecureMemory::recvReqRetry

Let's go ahead and declare and define `recvReqRetry` in the `public` scope of `SecureMemory`. Add the following lines to `inspector_gadget.hh` to declare `InpsectorGadget::recvReqRetry`:

```
  private:
    void recvReqRetry();
```

Now, let's define it. We simply need to try to schedule `nextReqSendEvent` for the `nextCycle`. Add the following code to `inspector_gadget.cc`:

```
void
SecureMemory::recvReqRetry()
{
    scheduleNextReqSendEvent(nextCycle());
}
```

Let's Do All of This for Response Path

So far, we have completed the functions required for the `request` path (from `cpuSidePort` to `memSidePort`). Now we have to do all of that for the `response` path. I'm not going to go over the details of that since they are going to look very similar to the functions for the `request` path.

However, here is a high level representation of both paths.

Request Path (without retries)

```
CPUSidePort.recvTimingReq
    ->SecureMemory.recvTimingReq
    ->SecureMemory.processNextReqSendEvent
    ->MemSidePort.sendPacket
```

Response Path (without retries)

```
MemSidePort.recvTimingResp
    ->SecureMemory.recvTimingResp
    ->SecureMemory.processNextRespSendEvent
    ->CPUSidePort.sendPacket
```

Response Path Additions to Header File

Let's declare the following in `inspector_gadget.hh` to implement the `response` path.

```
private:
  TimedQueue<PacketPtr> responseBuffer;

  EventFunctionWrapper nextRespSendEvent;
  void processNextRespSendEvent();
  void scheduleNextRespSendEvent(Tick when);

  EventFunctionWrapper nextRespRetryEvent;
  void processNextRespRetryEvent();
  void scheduleNextRespRetryEvent(Tick when);

public:
  bool recvTimingResp(PacketPtr pkt);
```

Defining Functions for the Response Path

Here is a comprehensive list of all the functions we need to declare and define for the `response` path. Let's not forget about `SecureMemory::recvRespRetry`.

```
bool MemSidePort::recvTimingResp(PacketPtr pkt);
void CPUSidePort::sendPacket(PacketPtr pkt);
void CPUSidePort::recvRespRetry();
bool SecureMemory::recvTimingResp(PacketPtr pkt);
void SecureMemory::recvRespRetry();
void SecureMemory::processNextRespSendEvent();
void SecureMemory::scheduleNextRespSendEvent(Tick when);
void SecureMemory::processNextRespRetryEvent();
void SecureMemory::scheduleNextRespSendEvent(Tick when);
```

To find the definition for all these functions please look at the complete version of `inspector_gadget.cc`. You can search for `Too-Much-Code` to find these functions.

SecureMemory::SecureMemory

Now, what we have to do is define the constructor of `SecureMemory`. To do it add the following code to `inspector_gadget.cc`:

```
SecureMemory::SecureMemory(const SecureMemoryParams& params):
    ClockedObject(params),
    cpuSidePort(this, name() + ".cpu_side_port"),
    memSidePort(this, name() + ".mem_side_port"),
    bufferEntries(params.inspection_buffer_entries),
    buffer(clockPeriod()),
    responseBufferEntries(params.response_buffer_entries),
    responseBuffer(clockPeriod()),
    nextReqSendEvent([this](){ processNextReqSendEvent(); }, name() + ".nextReqSendEvent"),
    nextReqRetryEvent([this](){ processNextReqRetryEvent(); }, name() + ".nextReqRetryEvent"),
    nextRespSendEvent([this](){ processNextRespSendEvent(); }, name() + ".nextRespSendEvent"),
    nextRespRetryEvent([this](){ processNextRespRetryEvent(); }, name() + ".nextRespRetryEvent")
{}
```

SimObject::init

Last step before compilation is to define the `init` function. Since `SecureMemory` is a `Responder` object, the convention is to let `peer` ports know that they can ask for their address range when the ranges become known. `init` is a `virtual` and `public` function from `SimObject`. Let's go ahead and declare it to override it. To do this, add the following declaration to the `public` scope of `SecureMemory` in `inspector-gadget.hh`.

```
virtual void init() override;
```

To define it, we need to simply call `sendRangeChange` from `cpuSidePort`. Add the following code to define `init` in `secure-memory.cc`

```
void
SecureMemory::init()
{
    cpuSidePort.sendRangeChange();
}
```

Let's Compile

We're ready to compile gem5. Let's do it and while we wait we will work on the configuration scripts. Run the following command in the base gem5 directory to rebuild gem5.

```
cd /workspaces/2024/gem5
scons build/NULL/gem5.opt -j$(nproc)
```

Let's Create a Configuration Script

For this step, we're going to borrow some of the material from Running Things in gem5. We are specifically going to copy the materials for using *TrafficGenerators*. We are going to further expand that material by extending the *ChanneledMemory* class to put an `SecureMemory` right before the memory controller.

Run the following commands in the base gem5 directory to create a directory for our configurations and copy the materials needed:

```
cp -r ../materials/03-Developing-gem5-models/04-ports/step-1/configs/bootcamp configs/
```

SecureMemory

We will need to do the following to extend `ChanneledMemory`:

1: In `SecureMemory.__init__`, we should create an object of `SecureMemory` for every memory channel. Let's store all of them in `self.secure_widgets`. We need to remember to expose `inspection_buffer_entries` and `response_buffer_entries` from `SecureMemory` to the user. Make sure to also expose the input arguments of `ChanneledMemory.__init__`.
2: Override `incorporate_memory` from `ChanneledMemory`. First call `ChanneledMemory.incorporate_memory` and then connect the `mem_side_port` from each `SecureMemory` object to the `port` from one `MemCtrl` object.
3: Override `get_mem_ports` from `ChanneledMemory` to replace `port` from `MemCtrl` objects with `cpu_side_port` from `SecureMemory` objects.

SecureMemory: Code

This is what should be in `gem5/configs/bootcamp/secure-memory/components/inspected_memory.py`:

```python
from typing import Optional, Sequence, Tuple, Union, Type

from m5.objects import (
    AddrRange,
    DRAMInterface,
    SecureMemory,
    Port,
)

from gem5.components.boards.abstract_board import AbstractBoard
from gem5.components.memory.memory import ChanneledMemory
from gem5.utils.override import overrides

class SecureMemory(ChanneledMemory):
    def __init__(
        self,
        dram_interface_class: Type[DRAMInterface],
        num_channels: Union[int, str],
        interleaving_size: Union[int, str],
        size: Optional[str] = None,
        addr_mapping: Optional[str] = None,
        inspection_buffer_entries: int = 16,
        response_buffer_entries: int = 32,
    ) -> None:
```

```python
        super().__init__(
            dram_interface_class,
            num_channels,
            interleaving_size,
            size=size,
            addr_mapping=addr_mapping,
        )
        self.secure_widgets = [
            SecureMemory(
                inspection_buffer_entries=inspection_buffer_entries,
                response_buffer_entries=response_buffer_entries,
            )
            for _ in range(num_channels)
        ]

    @overrides(ChanneledMemory)
    def incorporate_memory(self, board: AbstractBoard) -> None:
        super().incorporate_memory(board)
        for inspector, ctrl in zip(self.secure_widgets, self.mem_ctrl):
            inspector.mem_side_port = ctrl.port

    @overrides(ChanneledMemory)
    def get_mem_ports(self) -> Sequence[Tuple[AddrRange, Port]]:
        return [
            (ctrl.dram.range, inspector.cpu_side_port)
            for ctrl, inspector in zip(self.mem_ctrl, self.secure_widgets)
        ]
```

first-secure-memory-example.py

Now, let's just simply add the following imports to `gem5/configs/bootcamp/secure-memory/first-secure-memory-example.py`:

```
from components.inspected_memory import SecureMemory
from m5.objects.DRAMInterface import DDR3_1600_8x8
```

Let's now create an object of `SecureMemory` with the following parameters.

```
memory = SecureMemory(
    dram_interface_class=DDR3_1600_8x8,
    num_channels=2,
    interleaving_size=128,
    size="1GiB",
)
```

Now, let's run the following command to simulate our configuration script.

```
./build/NULL/gem5.opt --debug-flags=SecureMemory configs/bootcamp/secure-memory/first-secure-memory-example.py
```

In the next slide, there is a recording of my terminal when running the command above.

Output: first-secure-memory-example.py

<script src="https://asciinema.org/a/9j5QCBXn5098Oa63FpEmoYvLK.js" id="asciicast-9j5QCBXn5098Oa63FpEmoYvLK" async data-rows=32></script>

Statistics

In this step, we see how to add statistics to our `SimObjects` so that we can measure things with them. For now let's add statistics to measure the following.

1- The sum of the queueing latency in `buffer` experienced by each `Packet`. Let's use the name `totalbufferLatency` for this statistic.
2- Total number of `requests` forwarded. Let'use the name `numRequestsFwded`.
3- The sum of the queueing latency in `responseBuffer` experienced by each `Packet`. Let's use the name `totalResponseBufferLatency` for this statistic.
4- Total number of `requests` forwarded. Let'use the name `numResponsesFwded`.

Statistics:: Header File

gem5 has its own internal classes for measuring statistics (stats for short). Let's go ahead and include the header files for them in `src/bootcamp/inspector-gadget.hh`

```
#include "base/statistics.hh"
#include "base/stats/group.hh"
```

gem5 stats have multiple types, each useful for measuring specific types of data. We will look at using `statistics::Scalar` stats since all the things we want to measure are scalars.

Let's go ahead a declare a new `struct` called `SecureMemoryStats` inside the `private` scope of `SecureMemory` and also declare an instance of it. It will inherit from `statistics::Group`. Add the following lines to `src/bootcamp/inspector-gadget.hh` to do this.

```
  private:
    struct SecureMemoryStats: public statistics::Group
    {
        statistics::Scalar totalbufferLatency;
        statistics::Scalar numRequestsFwded;
        statistics::Scalar totalResponseBufferLatency;
        statistics::Scalar numResponsesFwded;
        SecureMemoryStats(SecureMemory* inspector_gadget);
    };
    SecureMemoryStats stats;
```

Statistics: Source File

Let's define the constructor of `SecureMemoryStats`. Add the following code under `namespace gem5` to do this.

```
SecureMemory::SecureMemoryStats::SecureMemoryStats(SecureMemory* inspector_gadget):
    statistics::Group(inspector_gadget),
    ADD_STAT(totalbufferLatency, statistics::units::Tick::get(), "Total inspection buffer latency."),
    ADD_STAT(numRequestsFwded, statistics::units::Count::get(), "Number of requests forwarded."),
    ADD_STAT(totalResponseBufferLatency, statistics::units::Tick::get(), "Total response buffer latency."),
    ADD_STAT(numResponsesFwded, statistics::units::Count::get(), "Number of responses forwarded.")
{}
```

Few things to note:

1- Initialize our stat object by adding `stats(this)` to the initialization list in the constructor `InspectorGdaget`.
2- `statistics::Group::Group` takes a pointer to an object of `statistics::Group` that will be its parent. Class `SimObject` inherits from `statistics::Group` so we can use a pointer to `SecureMemory` as that input.
3- The macro `ADD_STAT` registers and initializes our statistics that we have defined under the struct. The order of arguments are `name`, `unit`, `description`. To rid yourself of any headache, make sure the order of `ADD_STAT` macros match that of statistic declaration.

Counting Things

Now let's go ahead and start counting things with stats. We can simply count `numRequestsFwded` and `numResponsesFwded` in `processNextReqSendEvent` and `processNextRespSendEvent` respectively.

To do it, simply add the following lines inside the body of those functions.

```
void
SecureMemory::processNextReqSendEvent()
{
    // ...
    stats.numRequestsFwded++;
    PacketPtr pkt = buffer.front();
}

void
SecureMemory::processNextReqSendEvent()
{
    // ...
    stats.numResponsesFwded++;
    PacketPtr pkt = responseBuffer.front();
}
```

Measuring Queueing Latencies

To measure the queueing latency in `buffer` and `responseBuffer` we need to track the time each `Packet` is inserted in these buffers as well the time they are removed. We already track the insertion time for each `Packet`. We only need to make it accessible from the outside. We can use `curTick()` in `processNextReqSendEvent` and `processNextRespSendEvent` to track the time each `Packet` is removed from `buffer` and `responseBuffer` respectively.

Let's go ahead an add the following function inside the `public` scope of `TimedQueue`.

```
public:
  Tick frontTime() { return insertionTimes.front(); }
```

Measuring Queueing Latencies cont.

This is how `processNextReqSendEvent`, `processNextRespSendEvent` would look for measuring all statistics.

```
void
SecureMemory::processNextReqSendEvent()
{
    // ...
    stats.numRequestsFwded++;
    stats.totalbufferLatency += curTick() - buffer.frontTime();
    PacketPtr pkt = buffer.front();
}

void
SecureMemory::processNextReqSendEvent()
{
    // ...
    stats.numResponsesFwded++;
    stats.totalResponseBufferLatency += curTick() - responseBuffer.frontTime();
    PacketPtr pkt = responseBuffer.front();
}
```

Let's Rebuild and Simulate

We're ready to compile gem5. Let's do it and while we wait we will work on the configuration scripts. Run the following command in the base gem5 directory to rebuild gem5.

```
scons build/NULL/gem5.opt -j$(nproc)
```

Now, let's go ahead and run the simulation again. We don't need to make any changes to our configuration script. Run the following command in the base gem5 directory to run the simulation.

```
./build/NULL/gem5.opt configs/bootcamp/secure-memory/first-secure-memory-example.py
```

Now if you search for the name of the stats we added in `m5out/stats.txt`. This is what we will see. **NOTE**: I did by searching for the name of the `SecureMemory` objects in the file using `grep secure_widgets m5out/stats.txt` in the base gem5 directory.

```
system.memory.secure_widgets0.totalbufferLatency         7334                         # Total inspection buffer latency. (Tick)
system.memory.secure_widgets0.numRequestsFwded           22                         # Number of requests forwarded. (Count)
system.memory.secure_widgets0.totalResponseBufferLatency         8608                         # Total response buffer latency. (Tick)
system.memory.secure_widgets0.numResponsesFwded           22                         # Number of responses forwarded. (Count)
system.memory.secure_widgets0.power_state.pwrStateResidencyTicks::UNDEFINED   1000000000                       # Cumulative time (in ticks) in various power states (Tick)
system.memory.secure_widgets1.totalbufferLatency         6003                         # Total inspection buffer latency. (Tick)
system.memory.secure_widgets1.numRequestsFwded           18                         # Number of requests forwarded. (Count)
system.memory.secure_widgets1.totalResponseBufferLatency         7746                         # Total response buffer latency. (Tick)
system.memory.secure_widgets1.numResponsesFwded           18                         # Number of responses forwarded. (Count)
```

End of Step 1

Step 2: Walking the Merkle Tree

Now, we will extend the `SecureMemory` class to include walking a Merkle Tree.

For each packet that goes into the buffer, we will calculate the hash of the packet and store it in a Merkle Tree. When the packet is sent to the memory controller, we will calculate the hash of the packet and compare it with the hash stored in the Merkle Tree. If the hashes match, we will forward the packet to the memory controller. If the hashes do not match, we will drop the packet.

We will send extra memory accesses to the memory controller to read and update the Merkle Tree. We will also add statistics to measure the number of packets dropped and the number of memory accesses to the Merkle Tree.