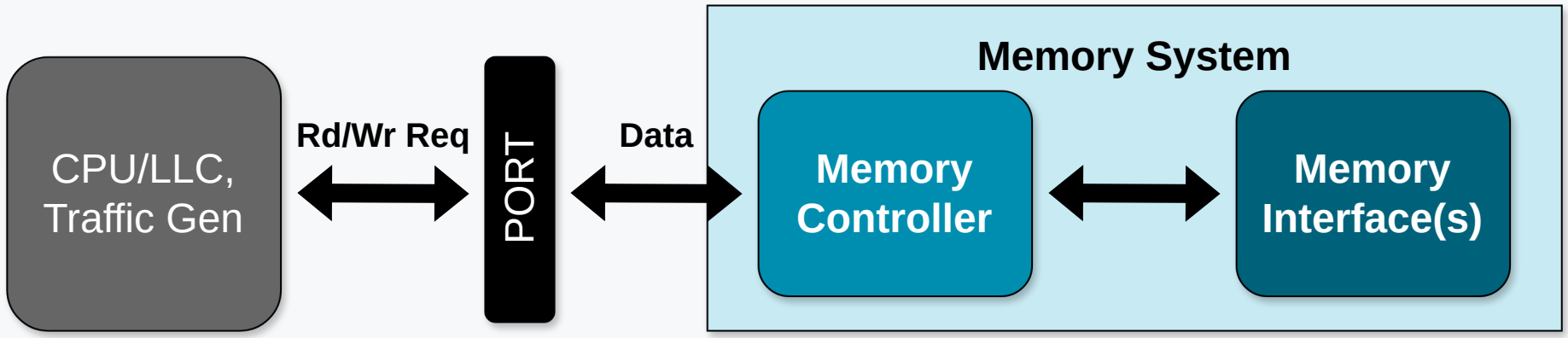
Modeling Memory in gem5

DRAM and other memory devices!

Memory System

gem5's memory system consists of two main components

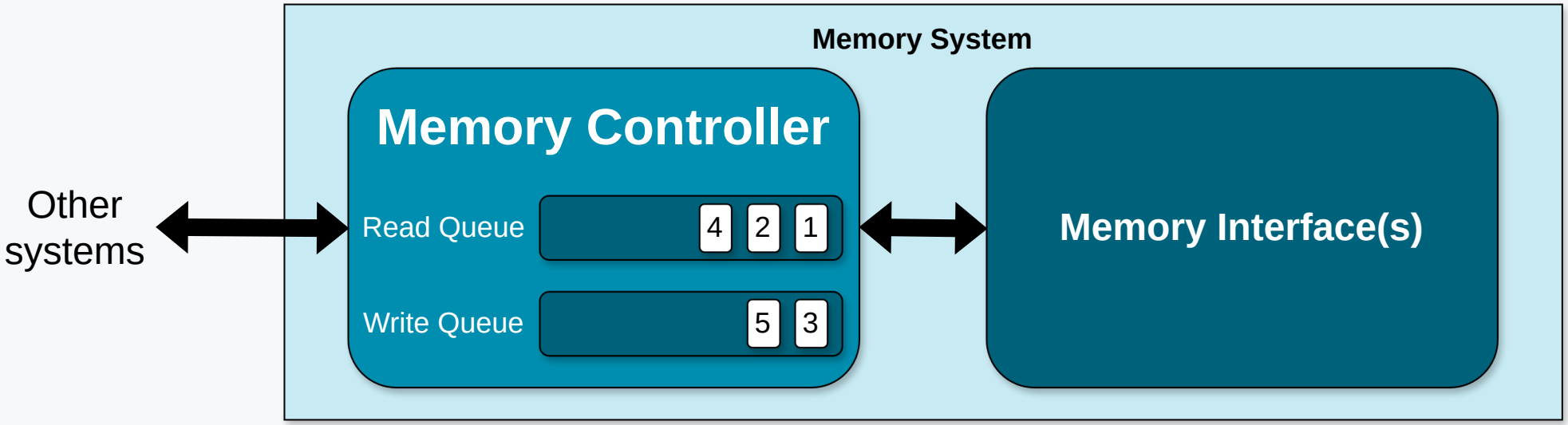
1. Memory Controller
2. Memory Interface(s)



Memory Controller

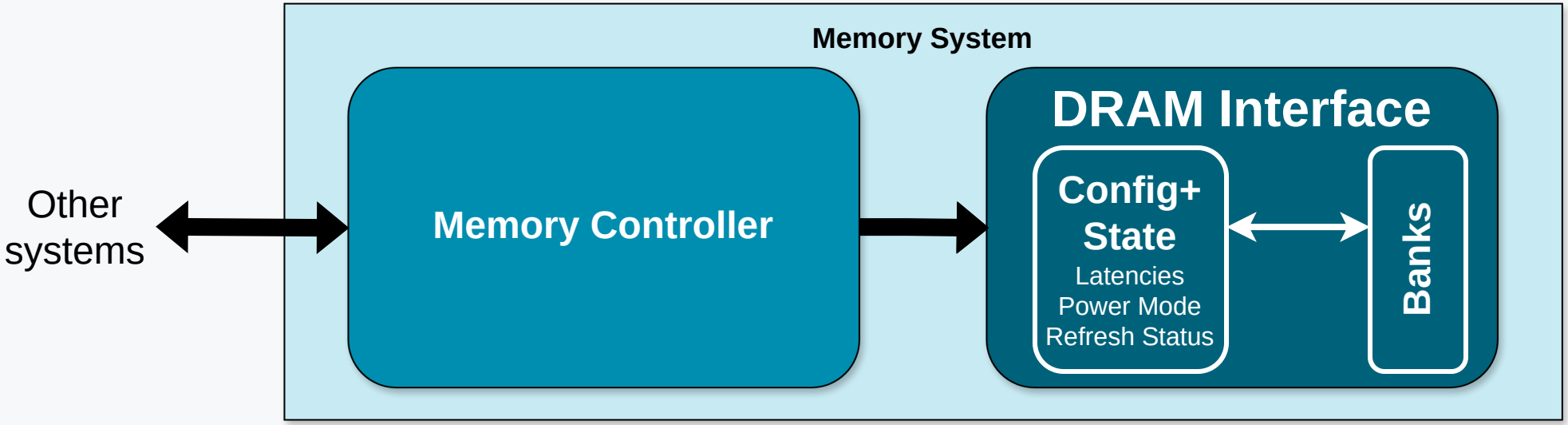
When `MemCtrl` receives packets...

1. Packets enqueued into the read and/or write queues
2. Applies **scheduling algorithm** (FCFS, FR-FCFS, ...) to issue read and write requests

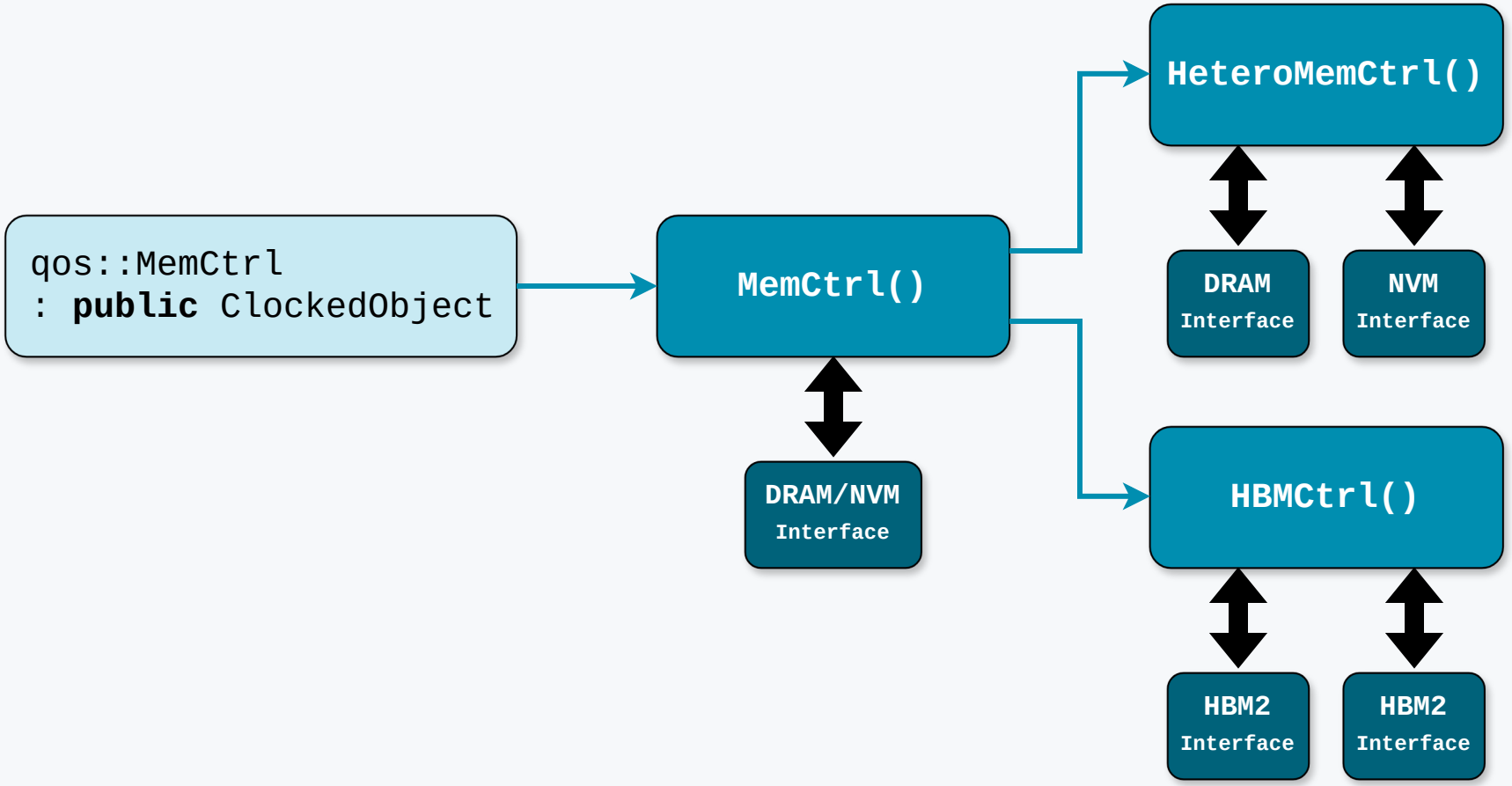


Memory Interface

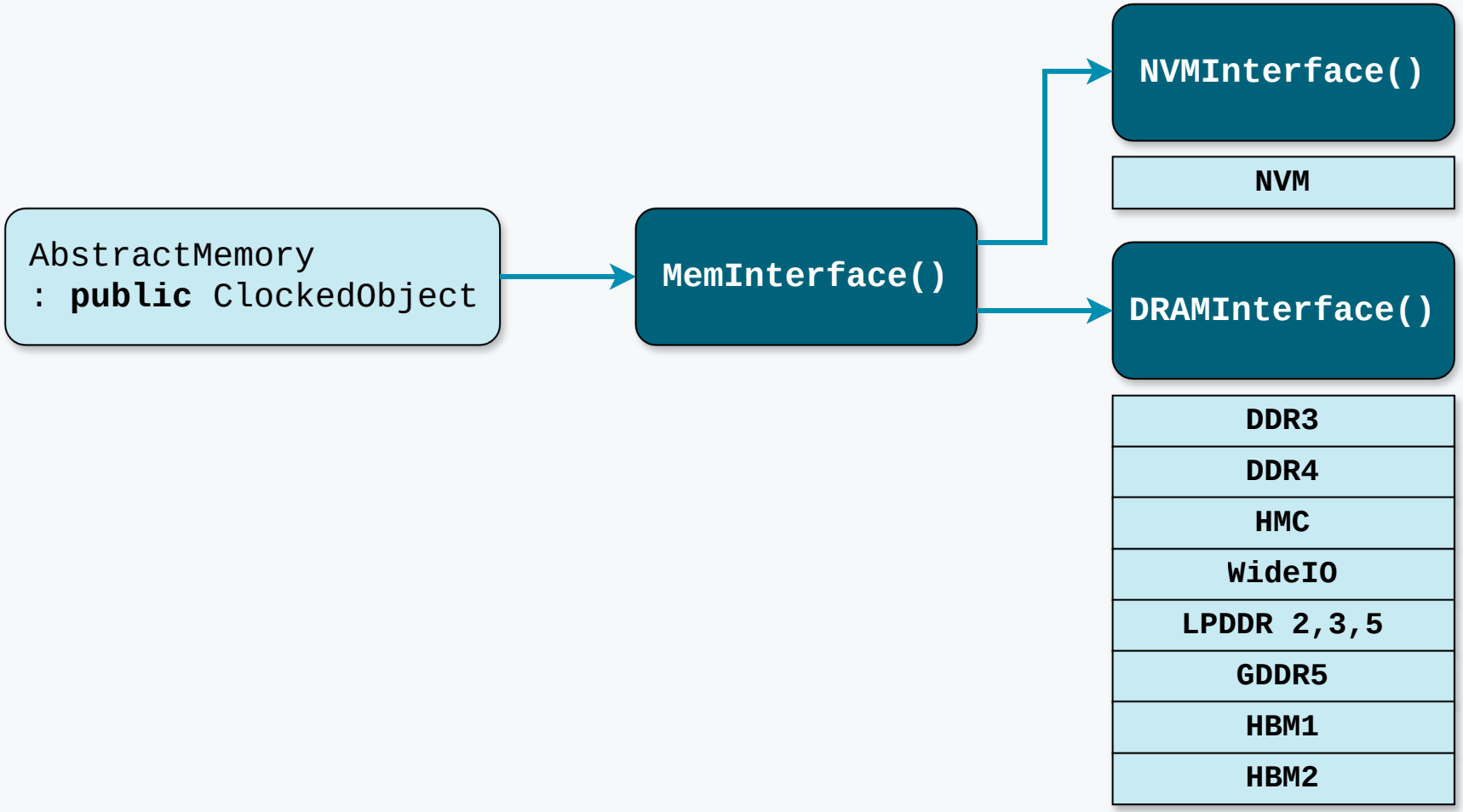
- The memory interface implements the **architecture** and **timing parameters** of the chosen memory type.
- It manages the **media specific operations** like activation, pre-charge, refresh and low-power modes, etc.



gem5's Memory Controllers



gem5's Memory Interfaces



How the memory model works

- The memory controller is responsible for scheduling and issuing read and write requests
- It obeys the timing parameters of the memory interface
  - `tCAS`, `tRAS`, etc. are tracked *per bank* in the memory interface
  - Use gem5 *events* (more later) to schedule when banks are free

The models isn't "cycle accurate," but it's *cycle level* and quite accurate compared to other DRAM simulators such as DRAMSim and DRAMSys.

You can extend the interface for new kinds of memory devices (e.g., DDR6), but usually you will use interfaces that have already been implemented.

The main way gem5's memory is normally configured is the number of channels and the channel/rank/bank/row/column bits since systems rarely use bespoke memory devices.

Memory in the standard library

The standard library wraps the DRAM/memory models into `MemorySystem`s.

Many examples are already implemented in the standard library for you.

See `gem5/src/python/gem5/components/memory/multi_channel.py` and `gem5/src/python/gem5/components/memory/single_channel.py` for examples.

Additionally,

- `SimpleMemory()` allows the user to not worry about timing parameters and instead, just give the desired latency. bandwidth, and latency variation
- `ChanneledMemory()` encompasses a whole memory system (both the controller and the interface)
- ChanneledMemory provides a simple way to use multiple memory channels
- ChanneledMemory handles things like scheduling policy and interleaving for you

Running an example with the standard library

Open `materials/02-Using-gem5/06-memory/run-mem.py`

This file uses traffic generators (seen previously) to generate memory traffic at 64GiB.

Let's see what happens when we use a simple memory. Add the following line for the memory system.

```
memory = SingleChannelSimpleMemory(latency="50ns", bandwidth="32GiB/s", size="8GiB", latency_var="10ns")
```

Run with the following. Use `-c <LinearGenerator,RandomGenerator>` to specify the traffic generators and `-r <read percentage>` to specify the percentage of reads.

```
gem5 run-mem.py
```

Vary the latency and bandwidth

Results for running with 16 GiB/s, 32 GiB/s, 64 GiB/s, and 100% reads and 50% reads.

Bandwidth	Read Percentage	Linear Speed (GB/s)	Random Speed (GB/s)
16 GiB/s	100%	17.180288	17.180288
	50%	17.180288	17.180288
32 GiB/s	100%	34.351296	34.351296
	50%	34.351296	34.351296
64 GiB/s	100%	34.351296	34.351296
	50%	34.351296	34.351296

With the `SimpleMemory` you don't see any complex behavior in the memory model (but it **is** fast).

Running Channeled Memory

- Open `gem5/src/python/gem5/components/memory/single_channel.py`
- We see `SingleChannel` memories such as:

```
def SingleChannelDDR4_2400(
    size: Optional[str] = None,
) -> AbstractMemorySystem:
    """
    A single channel memory system using DDR4_2400_8x8 based DIMM.
    """
    return ChanneledMemory(DDR4_2400_8x8, 1, 64, size=size)
```

- We see the `DRAMInterface=DDR4_2400_8x8`, the number of channels=1, interleaving_size=64, and the size.

Running Channeled Memory

- Lets go back to our script and replace the SingleChannelSimpleMemory with this!

Replace

```
SingleChannelSimpleMemory(latency="50ns", bandwidth="32GiB/s", size="8GiB", latency_var="10ns")
```

with

```
SingleChannelDDR4_2400()
```

Let's see what happens when we run our test

Vary the latency and bandwidth

Results for running with 16 GiB/s, 32 GiB/s, and 100% reads and 50% reads.

Bandwidth	Read Percentage	Linear Speed (GB/s)	Random Speed (GB/s)
16 GiB/s	100%	13.85856	14.557056
	50%	13.003904	13.811776
32 GiB/s	100%	13.85856	14.541312
	50%	13.058112	13.919488

As expected, because of read-to-write turn around, reading 100% is more efficient than 50% reads. Also as expected, the bandwidth is lower than the SimpleMemory (only about 75% utilization).

Somewhat surprising, the memory modeled has enough banks to handle random traffic efficiently.

Adding a new channeled memory

- Open `materials/02-Using-gem5/06-memory/lpddr2.py`
- If we wanted to add LPDDR2 as a new memory in the standard library, we first make sure there's a DRAM interface for it in the `dram_interfaces` directory
- Then we need to make sure we import it by adding the following to the top of your `lpddr2.py`:

```
from gem5.components.memory.abstract_memory_system import AbstractMemorySystem
from gem5.components.memory.dram_interfaces.lpddr2 import LPDDR2_S4_1066_1x32
from gem5.components.memory.memory import ChanneledMemory
```

Adding a new channeled memory

Then add the following to the body of `lpddr2.py`:

```
def SingleChannelLPDDR2(
    size: Optional[str] = None,
) -> AbstractMemorySystem:
    return ChanneledMemory(LPDDR2_S4_1066_1x32, 1, 64, size=size)
```

Then we import this new class to our script with:

```
from lpddr2 import SingleChannelLPDDR2
```

Let's test this again!

Vary the latency and bandwidth

Results for running with 16 GiB/s, and 100% reads and 50% reads.

Bandwidth	Read Percentage	Linear Speed (GB/s)	Random Speed (GB/s)
16 GiB/s	100%	4.089408	4.079552
	50%	3.65664	3.58816

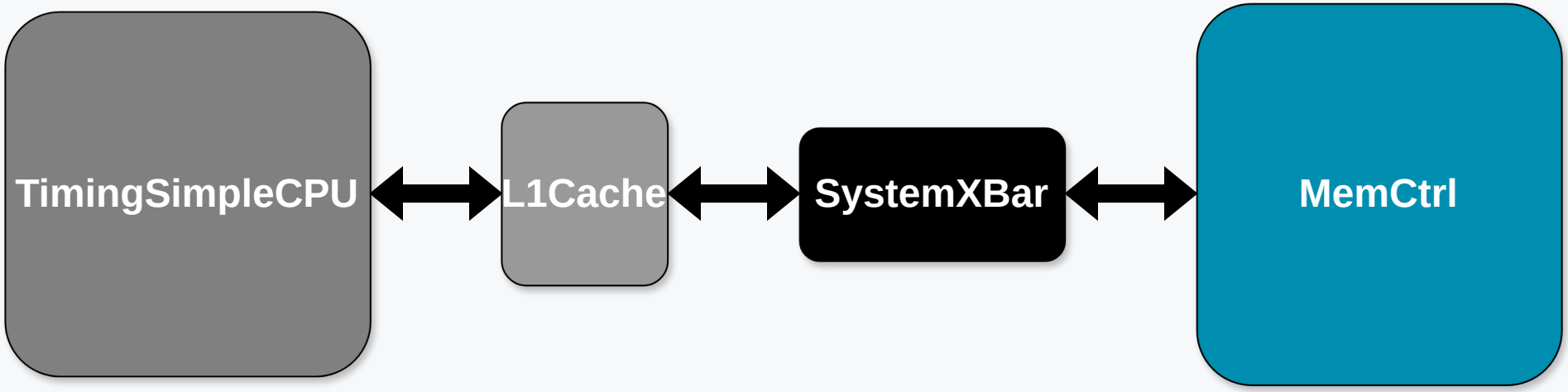
LPDDR2 doesn't perform as well as DDR4.

CommMonitor

- SimObject monitoring communication happening between two ports
- Does not have any effect on timing
- `gem5/src/mem/CommMonitor.py`

CommMonitor

Simple system to modify



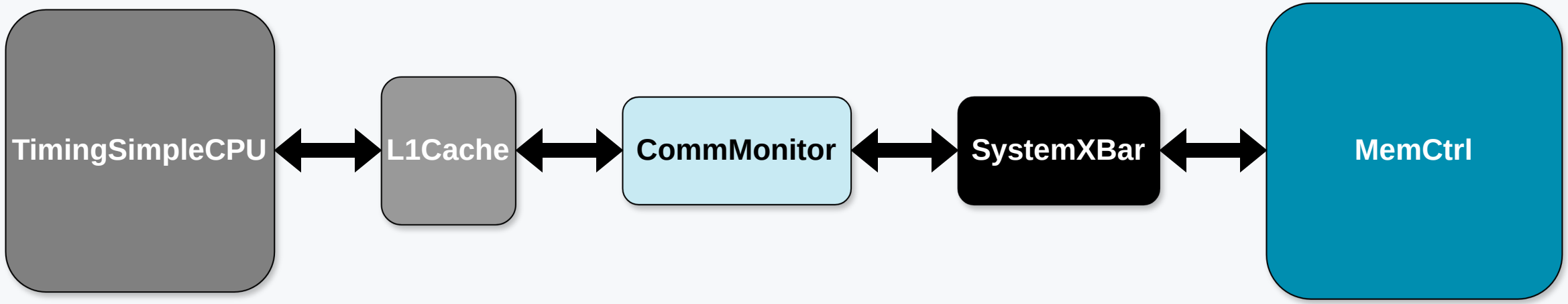
Let's simulate:

Run

```
gem5 comm_monitor.py
```

CommMonitor

Let's add the CommMonitor



CommMonitor

- Remove the line:

```
system.l1cache.mem_side = system.membus.cpu_side_ports
```

- Add the following block under the comment `# Insert CommMonitor here`:

```
system.comm_monitor = CommMonitor()
system.comm_monitor.cpu_side_port = system.l1cache.mem_side
system.comm_monitor.mem_side_port = system.membus.cpu_side_ports
```

- Run:

```
gem5 comm_monitor.py
```

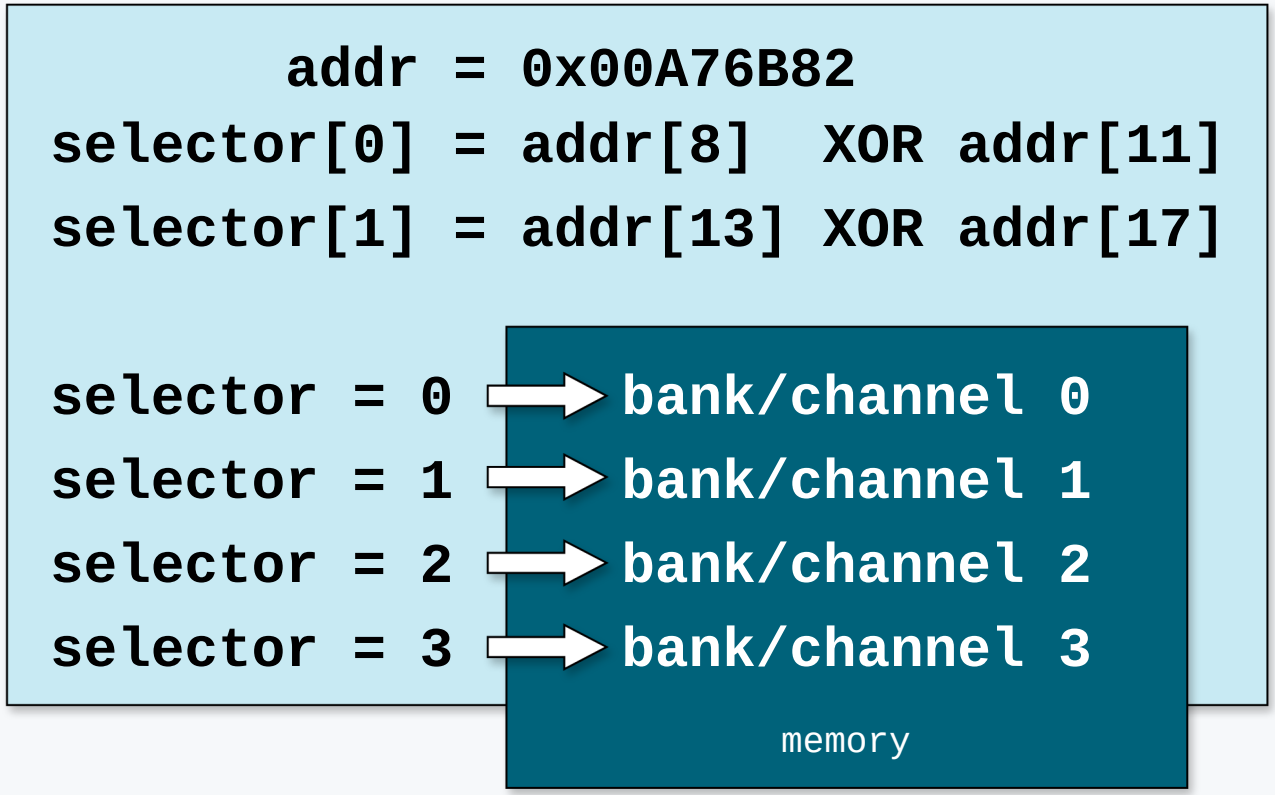
Address Interleaving

Idea: we can parallelize memory accesses

- For example, we can access multiple banks/channels/etc at the same time
- Use part of the address as a selector to choose which bank/channel to access
- Allows contiguous address ranges to interleave between banks/channels

Address Interleaving

For example...



Address Interleaving

Using address interleaving in gem5

- We can use AddrRange constructors to define a selector function
  - `src/base/addr_range.hh`
- Example: standard library's multi-channel memory
  - `gem5/src/python/gem5/components/memory/multi_channel.py`

Address Interleaving

There are two constructors

Constructor 1:

```
AddrRange(Addr _start,
          Addr _end,
          const std::vector<Addr> &_masks,
          uint8_t _intlv_match)
```

`_masks`: an array of masks, where bit `k` of selector is the XOR of all bits specified by `masks[k]`

Address Interleaving

There are two constructors

Constructor 2 (legacy):

```
AddrRange(Addr _start,
          Addr _end,
          uint8_t _intlv_high_bit,
          uint8_t _xor_high_bit,
          uint8_t _intlv_bits,
          uint8_t _intlv_match)
```

Selector defined as two ranges:

```
addr[_intlv_high_bit:_intlv_low_bit] XOR addr[_xor_high_bit:_xor_low_bit]
```