Modeling Memory in gem5

DRAM and other memory devices!

TO DO: Improve the exercise and improve discussion of memory interleaving.

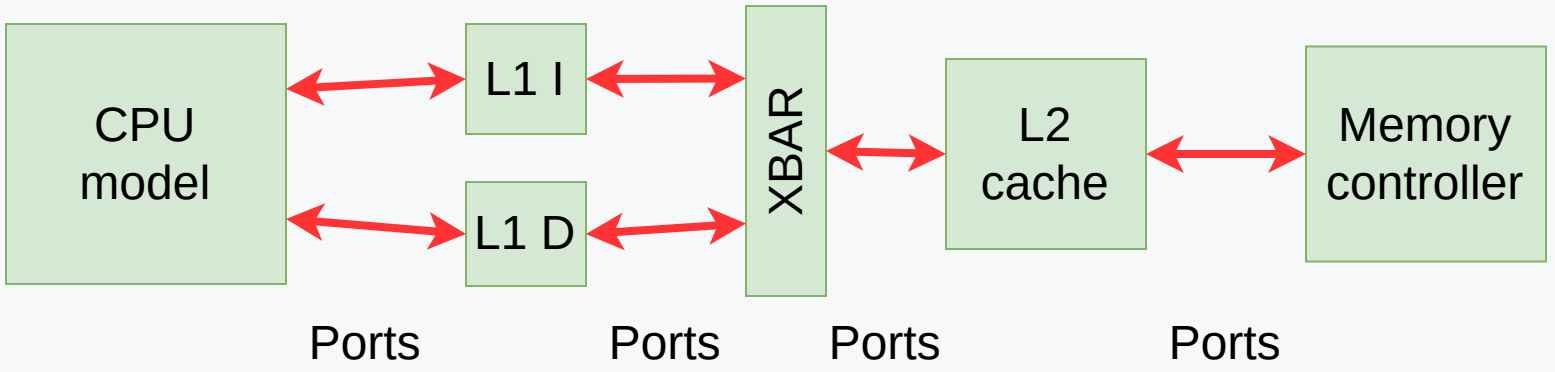
Reminder: gem5's software architecture

Ports are used to connect components in gem5. They are used to send and receive *packets* between components.

The memory controller has a *response port* which receives requests from something on the "cpu side" and sends responses.

The memory object has two jobs:

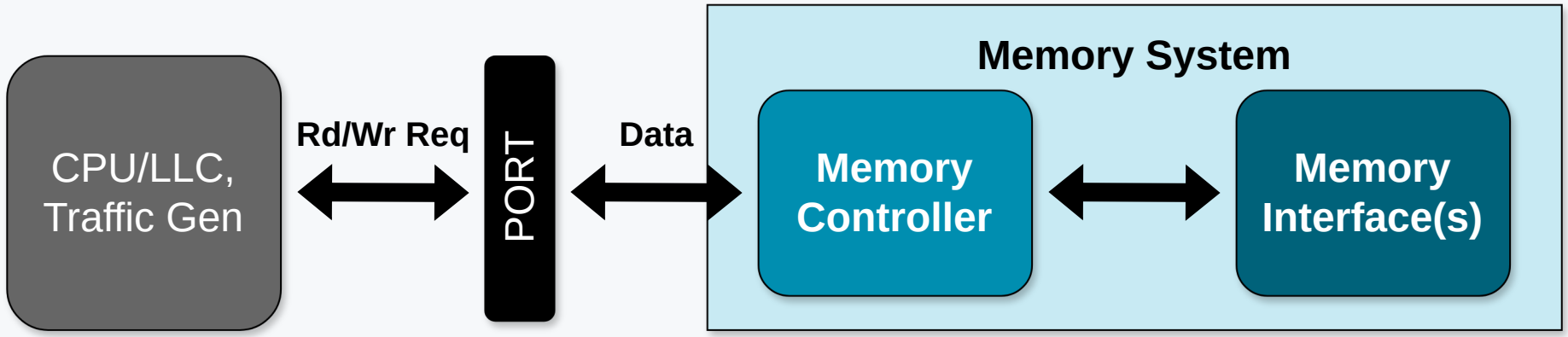
1. Model the functional memory data and respond with the correct data for the memory request.
2. Model the timing behavior of the memory device.



Memory System

gem5's memory system consists of two main components

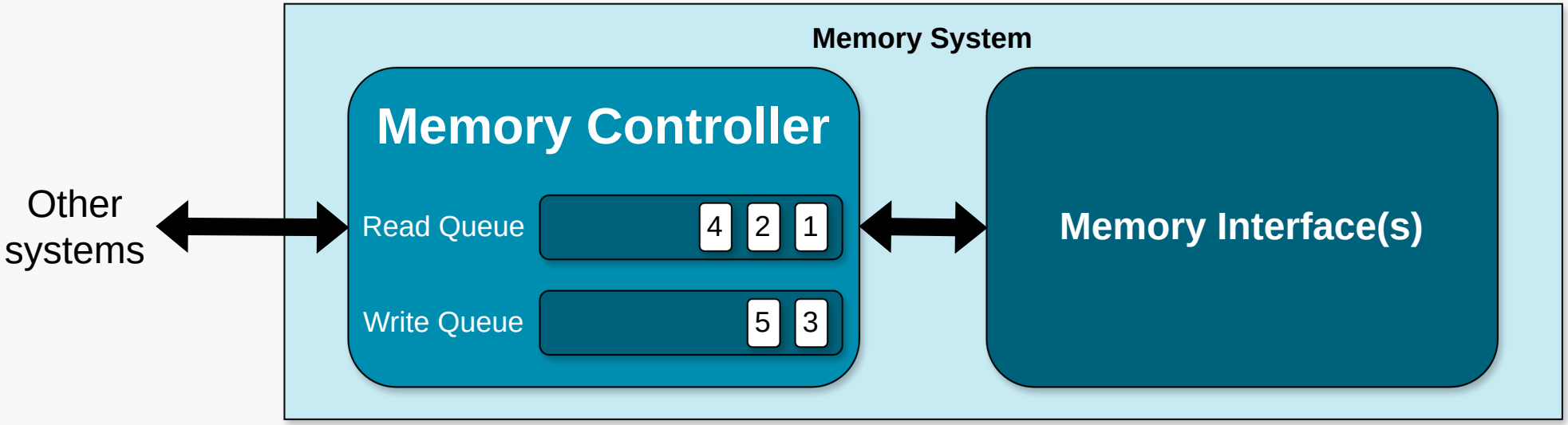
1. *Memory Controller*: Orders and schedules read and write requests
2. *Memory Interface(s)*: Implements the timing and architecture of the memory device



Memory Controller

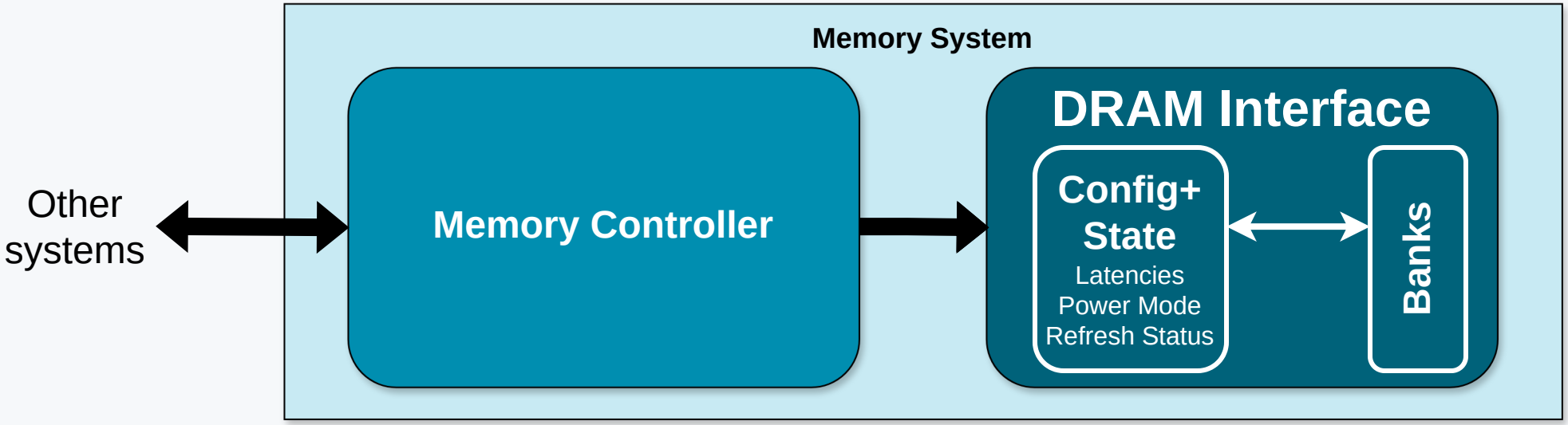
When `MemCtrl` receives packets...

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.



Included memory controllers

`MemCtrl`

This is the most common memory controller. Used for all DDR devices, LPDDR devices, and other DRAM devices.

`HeteroMemCtrl`

This memory controller allows you to have a heterogeneous memory system.
You can have both DRAM (e.g., DDR devices) and non-volatile memory (e.g., NVM devices) in the same system. This is like 3DXPoint systems where DRAM and NVM have separate memory spaces.

`HBMCtrl`

This is a controller specifically for HBM (High Bandwidth Memory) devices.
HBM needs its own controller because of the pseudo-channel architecture.
Each HBM controller actually has two different HBM interfaces to model the two pseudo-channels.

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 model 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.

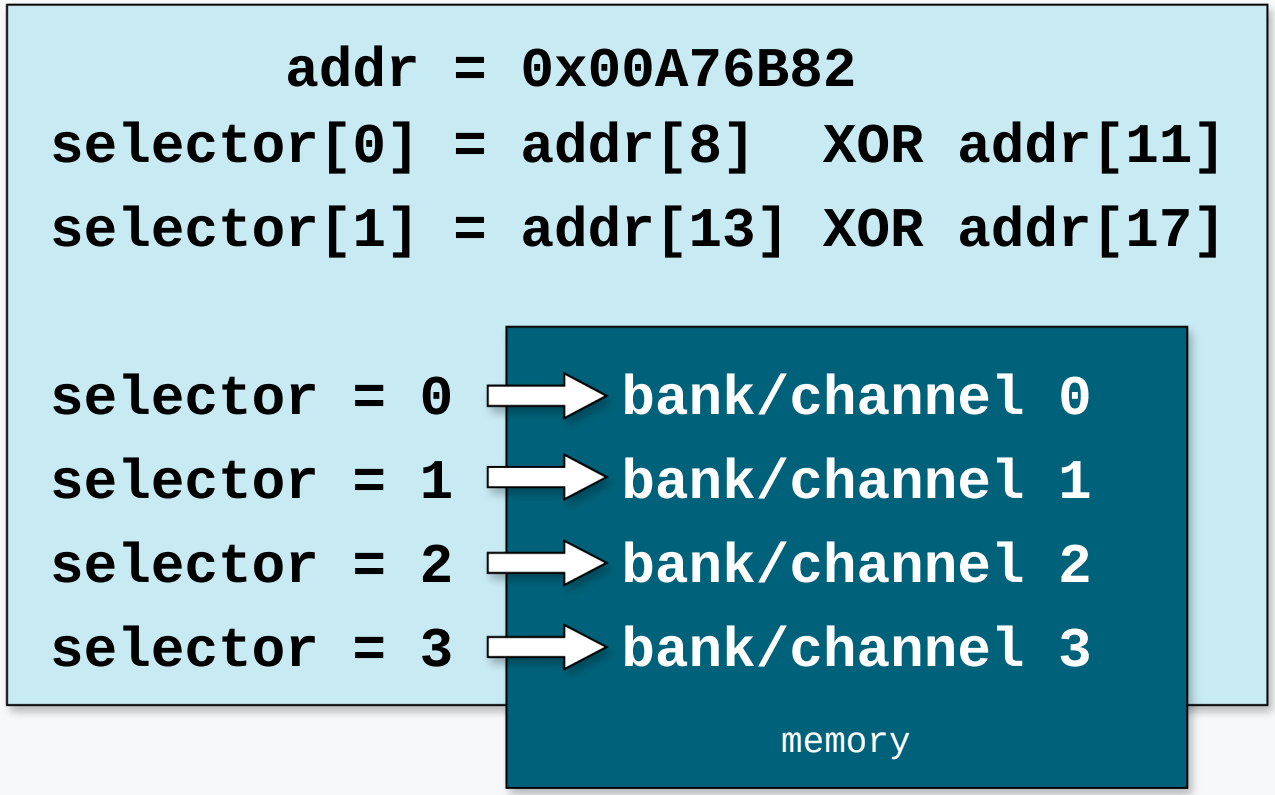
Address Interleaving

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`

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.

Using synthetic traffic generators

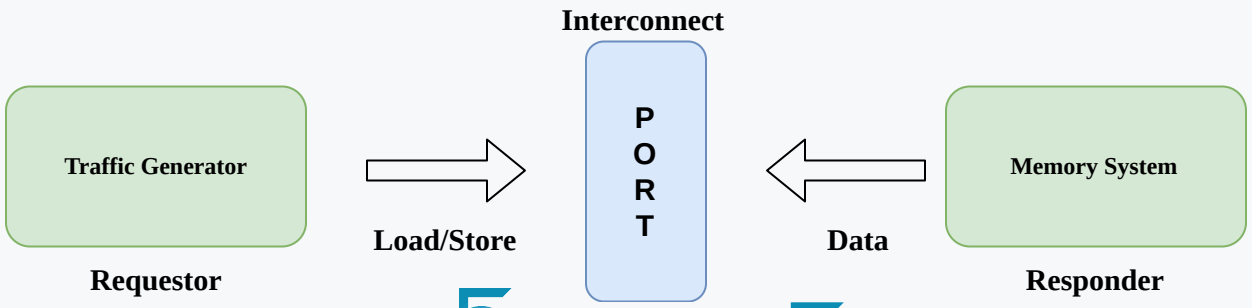
An aside on using synthetic traffic generators to test memory systems.

Synthetic Traffic Generation

Synthetic traffic generation is a technique for driven memory subsystems without requiring the simulation of processor models and running workload programs. We have to note the following about synthetic traffic generation.

- It can be used for the following: measuring maximum theoretical bandwidth, testing correctness of cache coherency protocol
- It can not be used for: measuring the execution time of workloads (even if you have their memory trace)

Synthetic traffic could follow a certain pattern like `sequential (linear)`, `strided`, and `random`. In this section we will look at tools in gem5 that facilitate synthetic traffic generation.



gem5: stdlib Components for Synthetic Traffic Generation

gem5's standard library has a collection of components for generating synthetic traffic. All such components inherit from `AbstractGenerator`, found in `src/python/gem5/components/processors`.

- These components simulate memory accesses. They are intended to replace a processor in a system that you configure with gem5.
- Examples of these components include `LinearGenerator` and `RandomGenerator`.

We will see how to use `LinearGenerator` and `RandomGenerator` to stimulate a memory subsystem. The memory subsystem that we are going to use is going to consist of a cache hierarchy with `private l1 caches and a shared l2 cache` with one channel of `DDR3` memory.

In the next slides we will look at `LinearGenerator` and `RandomGenerator` at a high level. We'll see how to write a configuration script that uses them.

LinearGenerator

Python Here

```
class LinearGenerator(AbstractGenerator):
    def __init__(
        self,
        num_cores: int = 1,
        duration: str = "1ms",
        rate: str = "100GB/s",
        block_size: int = 64,
        min_addr: int = 0,
        max_addr: int = 32768,
        rd_perc: int = 100,
        data_limit: int = 0,
    ) -> None:
```

RandomGenerator

Python Here

```
class RandomGenerator(AbstractGenerator):
    def __init__(
        self,
        num_cores: int = 1,
        duration: str = "1ms",
        rate: str = "100GB/s",
        block_size: int = 64,
        min_addr: int = 0,
        max_addr: int = 32768,
        rd_perc: int = 100,
        data_limit: int = 0,
    ) -> None:
```

LinearGenerator/RandomGenerator: Knobs

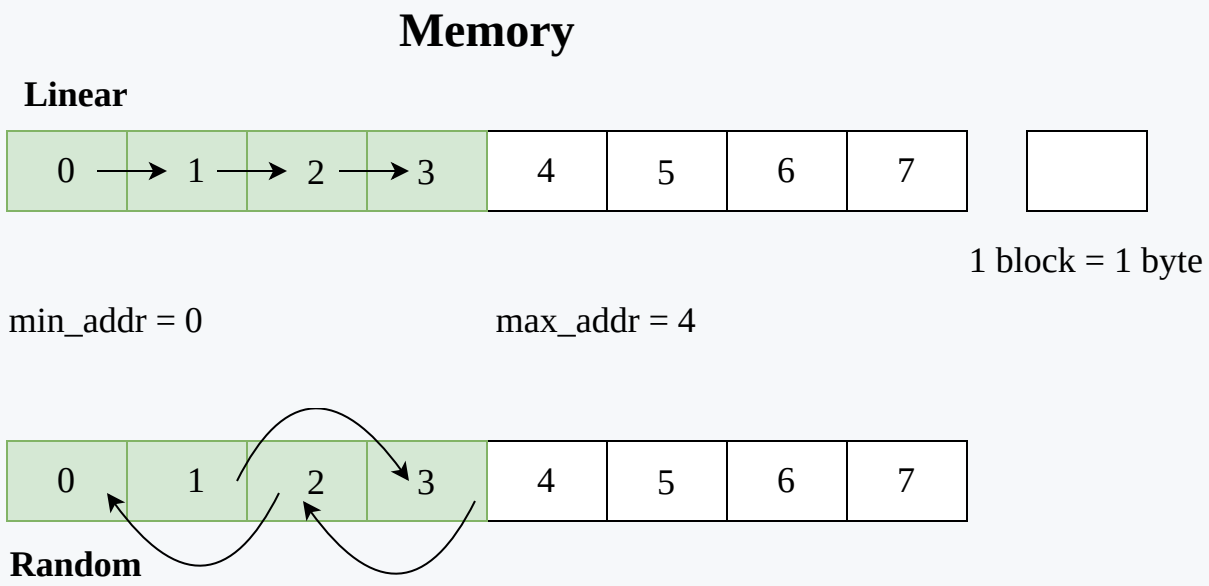
- **num_cores**
  - The number of cores in your system
- **duration**
  - Length of time to generate traffic
- **rate**
  - Rate at which to request data from memory
    - **Note**: This is *NOT* the rate at which memory will respond. This is the **maximum** rate at which requests will be made
- **block_size**
  - The number of bytes accessed with each read/write
- **min_addr**
  - The lowest memory address for the generator to access (via reads/writes)
- **max_addr**
  - The highest memory address for the generator to access (via reads/writes)
- **rd_perc**
  - The percentage of accesses that should be reads
- **data_limit**
  - The maximum number of bytes that the generator can access (via reads/writes)
    - **Note**: if `data_limit` is set to 0, there will be no data limit.

Traffic Patterns Visualized

`min_addr`: 0, `max_addr`: 4, `block_size`: 1

Linear: We want to access addresses 0 through 4 so a linear access would mean accessing memory in the following order.

Random: We want to access addresses 0 through 4 so a random access would mean accessing memory in any order. (In this example, we are showing the order: 1, 3, 2, 0).



Exercise: Measuring memory performance

In this exercise you will use different traffic patterns to better understand the performance characteristics of gem5's memory models.

Try to answer the questions before you run the experiments, then update your answers afterwards.

Questions

- When using the SimpleMemory model, how does the memory bandwidth change with different traffic patterns? Why or why not?
- When using DDR4 memory, how does the memory bandwidth change with different traffic patterns? Why or why not?
- Compare the performance of DDR4 to LPDDR2. Which has better bandwidth? What about latency? (Or, is this the wrong way to measure latency?)

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
from typing import Optional
```

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.