Using gem5's implementation of the CHI Protocol

Example

- Let's build a simple two-level cache hiearchy
  - Private L1 caches
  - Shared L2/directory (home node)
- Extend this to allow for multiple L2s (banked by address)
- Multiple memory controllers as well

Code in `materials/03-Developing-gem5-models/07-chi-protocol`.

Use some components

- There are some components already available for CHI
  - Just a `private_l1_moesi_cache` for now
  - Point-to-point network

See `gem5/src/python/gem5/components/cachehierarchies/chi/nodes/private_l1_moesi_cache.py`.

Create an L2 home node object

In CHI you have to specify many of the parameters to configure the cache.
We'll use the `AbstractNode` as the base class for our cache which hides some of the boilerplate.

For our L2, we want to parameterize just the size and the associativity. The other parameters are required for the `AbstractNode` class.

```
class SharedL2(AbstractNode):
    """A home node (HNF) with a shared cache"""

    def __init__(
        self,
        size: str,
        assoc: int,
        network: RubyNetwork,
        cache_line_size: int,
    ):
        super().__init__(network, cache_line_size)
```

Create the cache object

```
self.cache = RubyCache(
    size=size,
    assoc=assoc,
    # Can choose any replacement policy
    replacement_policy=RRIPRP(),
)
```

You can choose any replace policy from `gem5/src/mem/cache/replacement_policies/ReplacementPolicies.py`.

Set the CHI parameters

Set up home node that allows three hop protocols and enable the "owned" state.

```
self.is_HN = True
self.enable_DMT = True
self.enable_DCT = True
self.allow_SD = True
```

Set more CHI parameters

MOESI / Mostly inclusive for shared / Exclusive for unique

```
self.alloc_on_seq_acc = False
self.alloc_on_seq_line_write = False
self.alloc_on_readshared = True
self.alloc_on_readunique = False
self.alloc_on_readonce = True
self.alloc_on_writeback = True
self.alloc_on_atomic = True
self.dealloc_on_unique = True
self.dealloc_on_shared = False
self.dealloc_backinv_unique = False
self.dealloc_backinv_shared = False
```

Now, let's create the hierarchy

Set the parameters we care about (and ignore others)

```
class PrivateL1SharedL2CacheHierarchy(AbstractRubyCacheHierarchy):
    """A two level cache based on CHI
    """

    def __init__(self, l1_size: str, l1_assoc: int, l2_size: str, l2_assoc: int):
        self._l1_size = l1_size
        self._l1_assoc = l1_assoc
        self._l2_size = l2_size
        self._l2_assoc = l2_assoc
```

Set up the hierarchy

Remember, `incorporate_cache` is the main method we need to implement. Much of the boilerplate is already available for you.

You should add code to create the shared L2 cache.

```python
def incorporate_cache(self, board):
    ...
    self.l2cache = SharedL2(
        size=self._l2_size,
        assoc=self._l2_assoc,
        network=self.ruby_system.network,
        cache_line_size=board.get_cache_line_size()
    )
    self.l2cache.ruby_system = self.ruby_system
    ...
```

Next, let's create the run script

First, let's use the traffic generator. Put the following code in `run_test.py`

```
from hierarchy import PrivateL1SharedL2CacheHierarchy

board = TestBoard(
    generator=LinearGenerator(num_cores=4, max_addr=2**22, rd_perc=75),
    cache_hierarchy=PrivateL1SharedL2CacheHierarchy(
        l1_size="32KiB", l1_assoc=8, l2_size="2MiB", l2_assoc=16,
    ),
    memory=SingleChannelDDR4_2400(size="2GB"),
    clk_freq="3GHz",
)
sim = Simulator(board)
sim.run()
```

Test the new hierarchy and look at the stats

```
> gem5-chi run_test.py
```

stats.txt

```
simSeconds                                   0.001000
...
board.processor.cores0.generator.readBW   2811101367.231156
board.processor.cores0.generator.writeBW  986163850.461362
board.processor.cores1.generator.readBW   2679838984.383712
board.processor.cores1.generator.writeBW  935348476.506769
board.processor.cores2.generator.readBW   2805533435.828071
board.processor.cores2.generator.writeBW  974899989.232133
board.processor.cores3.generator.readBW   2729054378.050062
board.processor.cores3.generator.writeBW  948724311.716480
```

Now, let's run a full system simulation

Let's create a script to run IS from NPB.
Just add the following to the template in `materials/03-Developing-gem5-models/07-chi-protocol/run-is.py`.

```
from hierarchy import PrivateL1SharedL2CacheHierarchy
cache_hierarchy = PrivateL1SharedL2CacheHierarchy(
    l1_size="32KiB",
    l1_assoc=8,
    l2_size="2MiB",
    l2_assoc=16,
)
```

Run the script

```
gem5 run-is.py
```

You should see the following output pretty quickly

```
...
Work begin. Switching to detailed CPU
switching cpus
...
```

This takes about 5 minutes to complete, but you can check the output while it's running with `tail -f m5out/board.pc.com_1.terminal`.

Grab some stats

Finally, let's grab some stats that seem interesting (we'll use these more in the next section).

```
board.cache_hierarchy.ruby_system.m_missLatencyHistSeqr::mean   185.561335
board.processor.switch0.core.commitStats0.ipc     0.149605
```

We have an average miss latency of 185 cycles (lots of L2 misses!) and an IPC of 0.15.

Note: This example has not been debugged and may have FS issues

Summary

- We've created a simple two-level cache hierarchy using the CHI protocol
- We've run a simple traffic generator and a full system simulation
- We've seen how to set up the CHI protocol in gem5 with the standard library