Modeling CPU cores in gem5

gem5 simulator

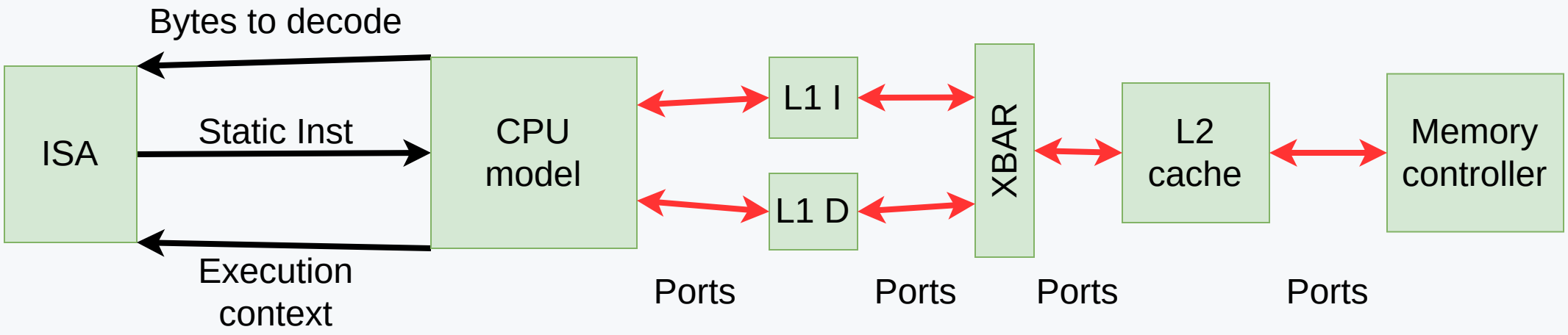
Remember, gem5 is an execute-in-execute simulator.

We have looked at what happens to memory requests going through the caches and memory system.

Now, let's see how applications and their instructions are executed in gem5.

Executing an instruction

- **Fetch**: Get the next instruction from memory using the instruction cache port
  - Possibly checks the branch predictor to see if the instruction is a branch
- **Decode**: Calls into the ISA to decode the instruction and get an instance of the `StaticInst` class
- **Execute**: Calls into the `ExecContext` instance to execute the instruction. Actually update the registers
- **Memory**: If the instruction needs to access memory, it uses a different path than execute to send request to the data cache



Aside: Predicting instruction execution flow

Instructions are usually executed in sequential order i.e. one after another. However, some instructions allow the program to jump, or *branch* to other parts of the program.

To ensure the hardware is ready to execute instructions—no matter the possible order—it is good to *predict* the direction of branching instructions.

Predicting the direction (e.g. a branch instruction is either "taken" or "not taken") of branch instructions is called **branch prediction**.

Aside: Branch Prediction

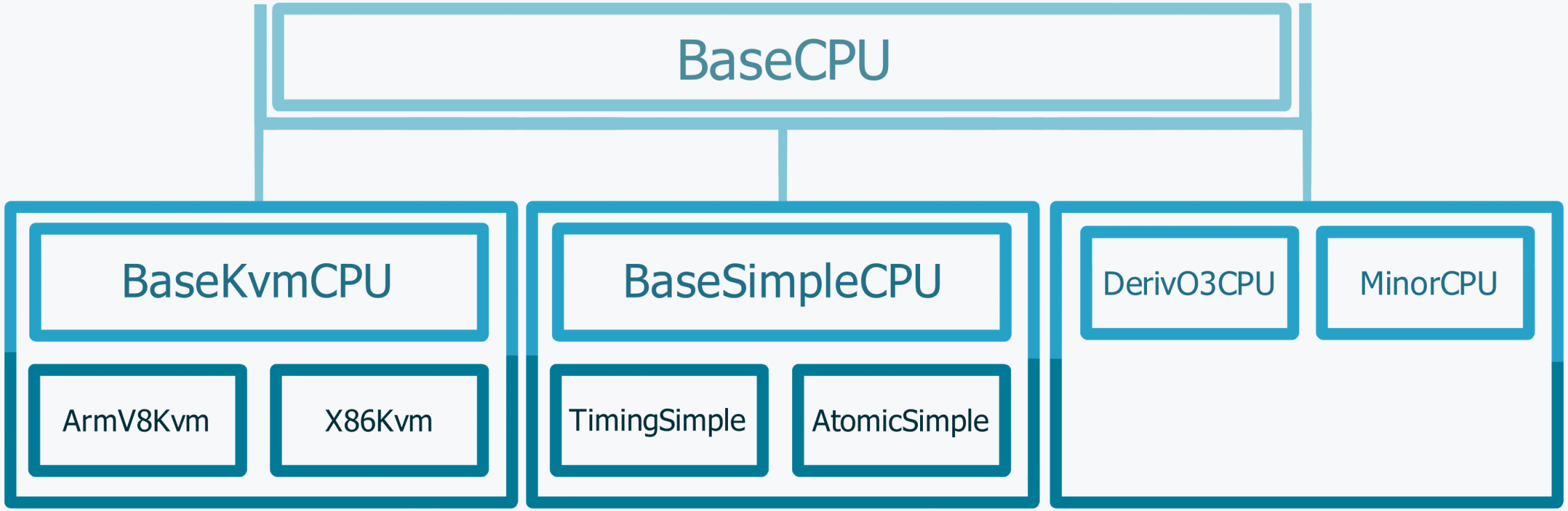
Better branch prediction leads to better performance, all other factors remaining same!

- Simplest predictors:
  - Just always predict "taken"! (or "not taken".)
  - Predic what the most recent direction was.
  - Similarly, keep a saturating counter which increases upon a "taken" and decreases on "not taken". Predict based on the value of the counter.
- State-of-the-art predictors:
  - TAGE-based and/or
  - Perceptron-based

Outline

- **Learn about CPU models in gem5**
  - AtomicSimpleCPU, TimingSimpleCPU, O3CPU, MinorCPU, KvmCPU
- Using the CPU models
  - Set up a simple system with two cache sizes and three CPU models
- Look at the gem5 generated statistics
  - To understand the differences between CPU models
- Create a custom processor
  - Change parameters of a processor based on O3CPU

gem5 CPU Models



SimpleCPU

Atomic

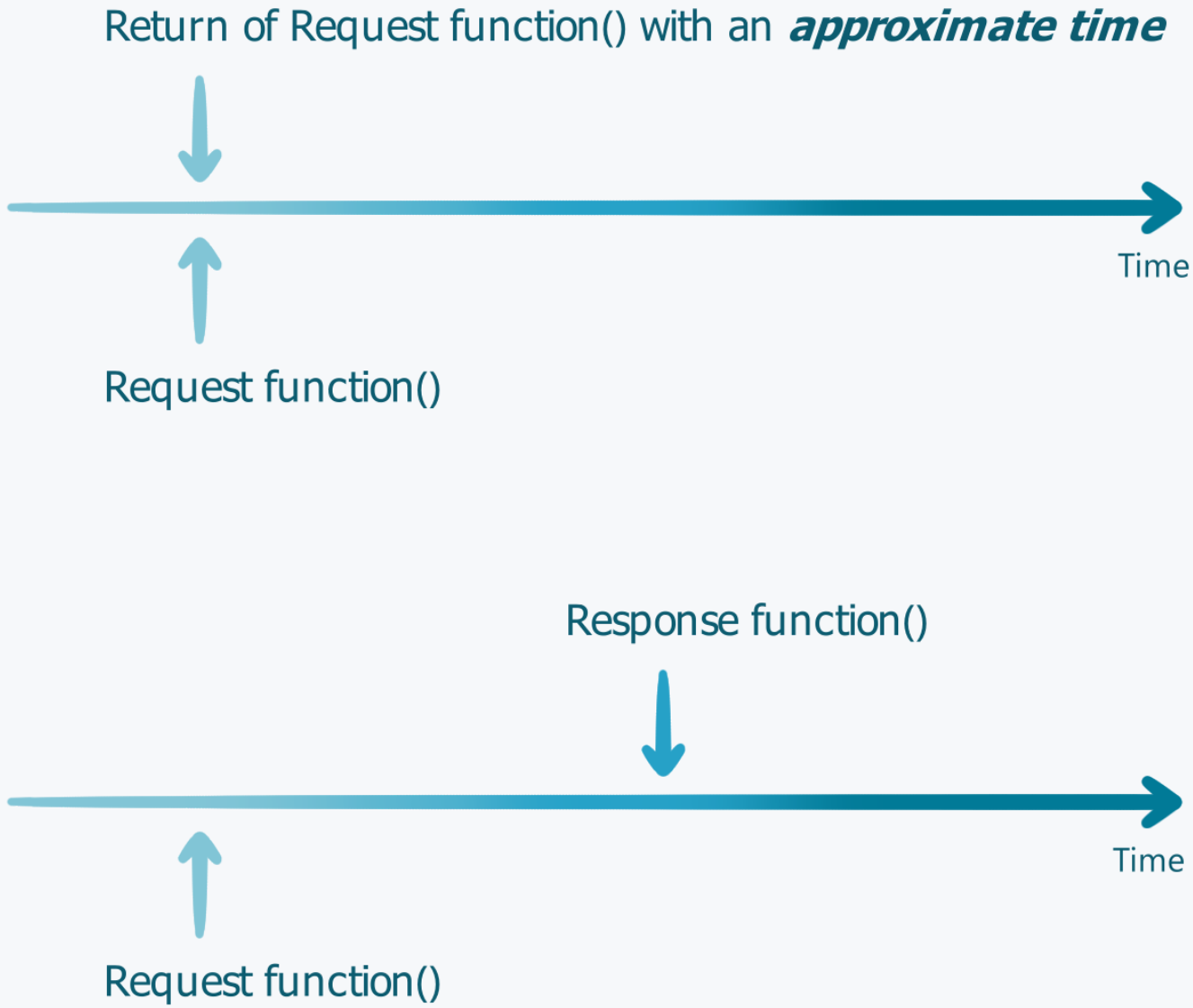
Sequence of nested calls
Use: Warming up, fast-forwarding

Functional

Backdoor access to mem.
(loading binaries)
No effect on coherency states

Timing

Split transactions
Models queuing delay and
resource contention



Other Simple CPUs

AtomicSimpleCPU

- Uses ***Atomic*** memory accesses
  - No resource contentions or queuing delay
  - Mostly used for fast-forwarding and warming of caches

TimingSimpleCPU

- Uses ***Timing*** memory accesses
  - Execute non-memory operations in one cycle
  - Models the timing of memory accesses in detail

O3CPU (Out of Order CPU Model)

- ***Timing*** memory accesses *execute-in-execute* semantics
- Time buffers between stages

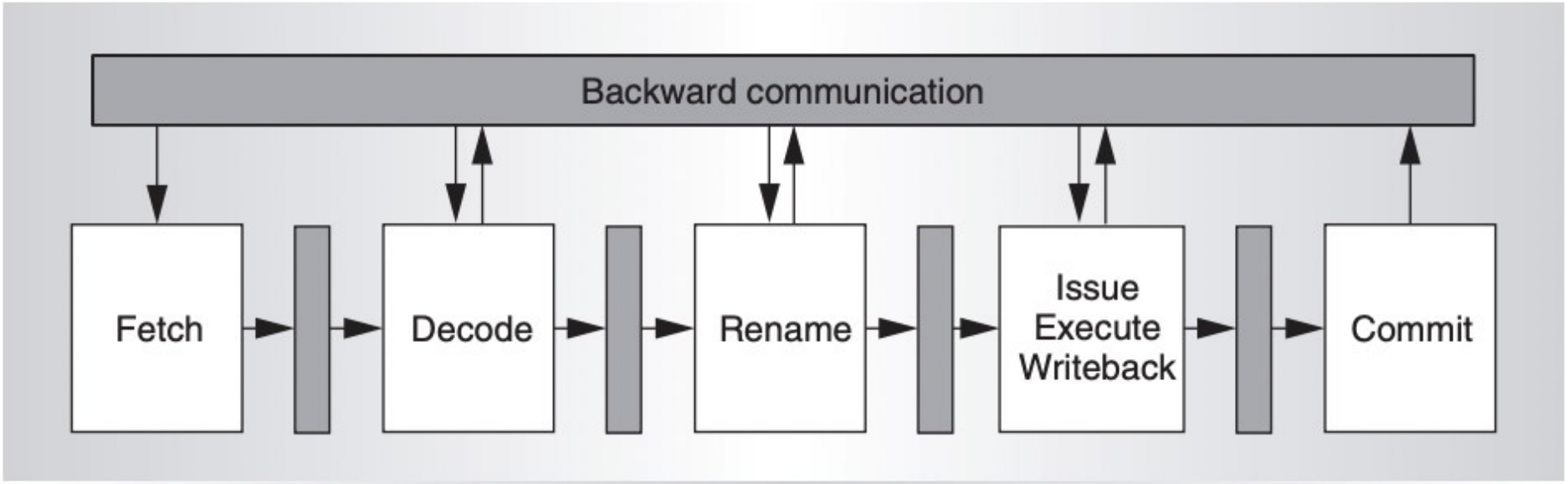


Figure 2. O3CPU pipeline. Shaded boxes represent time buffers.

The O3CPU Model has many parameters

src/cpu/o3/BaseO3CPU.py

```
decodeToFetchDelay = Param.Cycles(1, "Decode to fetch delay")
renameToFetchDelay = Param.Cycles(1, "Rename to fetch delay")
...
fetchWidth = Param.Unsigned(8, "Fetch width")
fetchBufferSize = Param.Unsigned(64, "Fetch buffer size in bytes")
fetchQueueSize = Param.Unsigned(
    32, "Fetch queue size in micro-ops per-thread"
)
...
```

Remember, do not update the parameters directly in the file. Instead, create a new *stdlib component* and extend the model with new values for parameters.
We will do this soon.

Minor CPU

- **Overview**: The Minor CPU model in gem5 is an in-order pipelined processor designed to help computer architects understand event-driven simulators.
- **Purpose**: The Minor CPU model provides a simplified yet detailed implementation of an in-order pipelined processor, making it easier to grasp the concepts of pipeline stages and their interactions.

Key Features:

- **In-Order Execution**: The Minor CPU executes instructions in the order they are fetched.
- **Pipeline Stages**: It consists of four main pipeline stages: Fetch1, Fetch2, Decode, and Execute.
- **SimObjects**: Each stage is implemented as a SimObject, allowing for modular and detailed simulation.

Minor CPU design

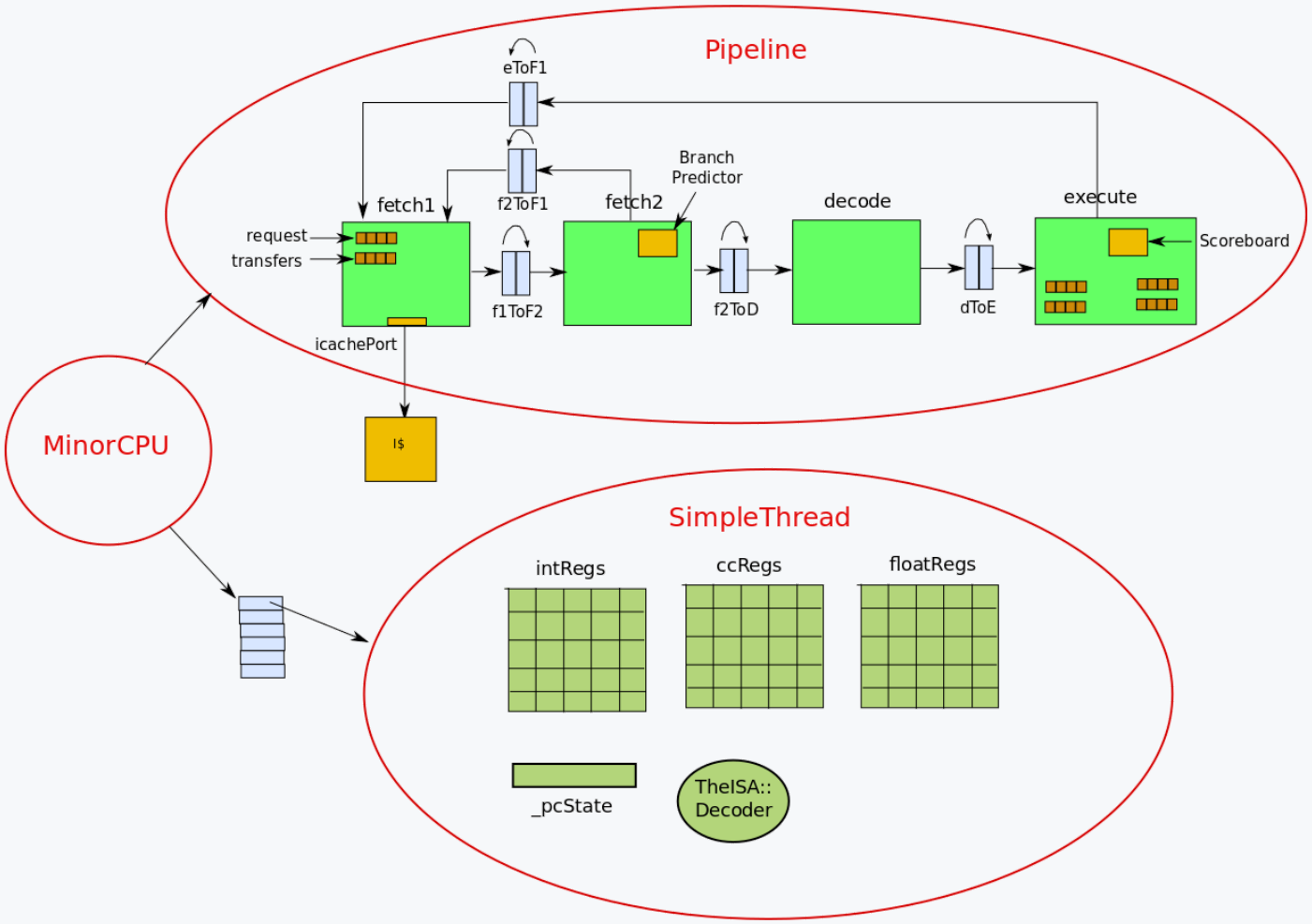
Pipeline Stages:

- **Fetch1**: Handles ITLB access and instruction fetch from main memory.
- **Fetch2**: Responsible for instruction decoding.
- **Decode**: Performs book-keeping tasks.
- **Execute**: Manages instruction issue, execution, memory access, writeback, and commit.

Implementation:

- **Latches**: The stages are connected via latches, which are used to transfer data between stages.
- **Evaluate Method**: Each stage has an `evaluate()` method that defines the actions performed at each CPU tick.

MinorCPU


Pipeline
eToF1
f2ToF1
Branch
Predictor
fetch1
fetch2
decode
execute
request
transfers
Scoreboard
f1ToF2
f2ToD
dToE
icachePort
I$
MinorCPU
SimpleThread
intRegs
ccRegs
floatRegs
_pcState
TheISA::
Decoder

Minor CPU parameters

Like the O3CPU, the Minor CPU model has many parameters that can be modified to change the behavior of the CPU.
src/cpu/minor/BaseMinorCPU.py

```
fetch1FetchLimit = Param.Unsigned(
    1, "Number of line fetches allowable in flight at once"
)
fetch1ToFetch2ForwardDelay = Param.Cycles(
    1, "Forward cycle delay from Fetch1 to Fetch2 (1 means next cycle)"
)
...
executeIssueLimit = Param.Unsigned(
    2, "Number of issuable instructions in Execute each cycle"
)
```

Customizing instruction performance

Both the O3CPU and Minor CPU models allow you to customize the performance of instructions by changing the parameters of the CPU model.

These models have functional units and pools of functional units.

- `MinorFU` and `MinorFUPool` for the Minor CPU
- `FUDesc` and `FUPool` for the O3CPU

The pool of functional units is what the CPU uses to execute instructions.
E.g., you could have an integer unit, a floating-point unit, and a memory unit.

Each unit has a "count" for the number of parallel units available and a list of operations that the unit can execute.

Each operation can be customized to execute a class of instructions and for each class of instructions it can have a latency and a throughput.

Example of a functional unit pool

From src/cpu/o3/FUPool.py and src/cpu/o3/FuncUnitConfig.py

```
class IntALU(FUDesc):
    opList = [OpDesc(opClass="IntAlu")]
    count = 6
class IntMultDiv(FUDesc):
    opList = [
        OpDesc(opClass="IntMult", opLat=3),
        OpDesc(opClass="IntDiv", opLat=20, pipelined=False),
    ]
    count = 2
...
class DefaultFUPool(FUPool):
    FUList = [IntALU(),IntMultDiv(),FP_ALU(),FP_MultDiv(),
        ReadPort(),SIMD_Unit(),PredALU(),WritePort(),
        RdWrPort(),IprPort(),
    ]
```

In this example, there are 6 parallel integer units which can execute all of the instructions tagged as "IntAlu". By default, they will complete in 1 cycle with a 1 cycle throughput.

The `IntMultDiv` unit has 2 parallel units, one of which can execute integer multiplication in 3 cycles and the other can execute integer division in 20 cycles.
For the multiply, a new instructions can start every cycle and for the division, the unit is not pipelined.

The pool has a list of all the functional units that are available to the CPU.

KvmCPU

- KVM – Kernel-based virtual machine
- Used for native execution on x86 and ARM host platforms
- Guest and the host need to have the same ISA
- Very useful for functional tests and fast-forwarding

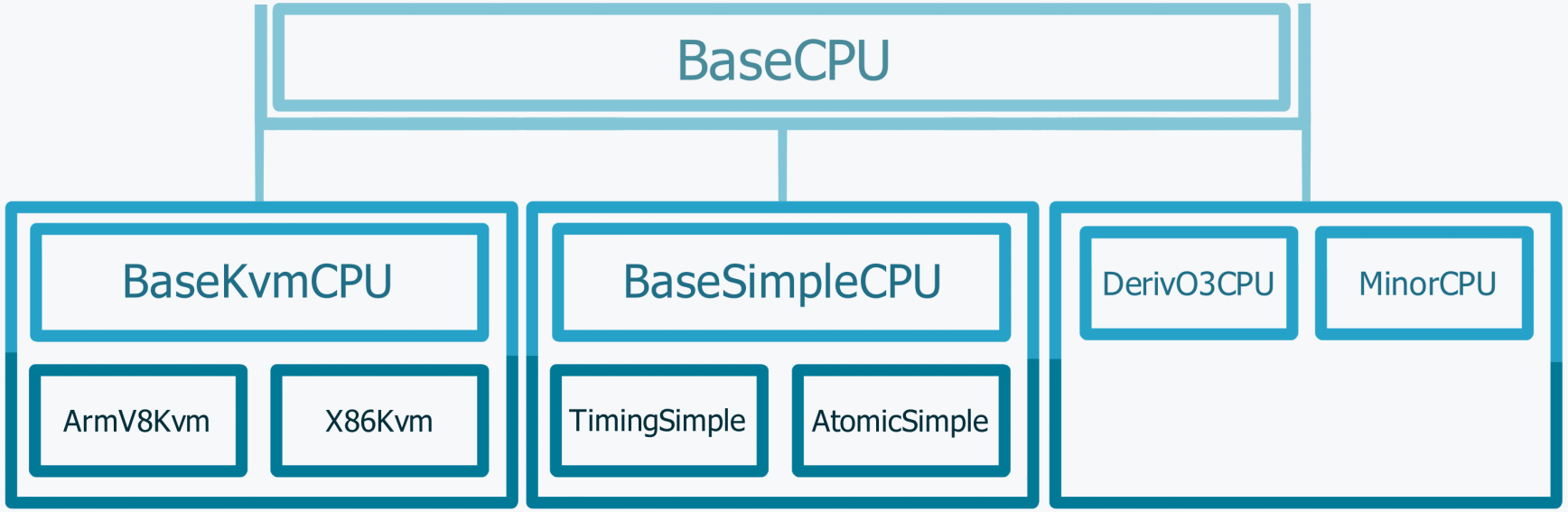
Summary of gem5 CPU Models

BaseKvmCPU

- Very fast
- No timing
- No caches, BP

BaseSimpleCPU

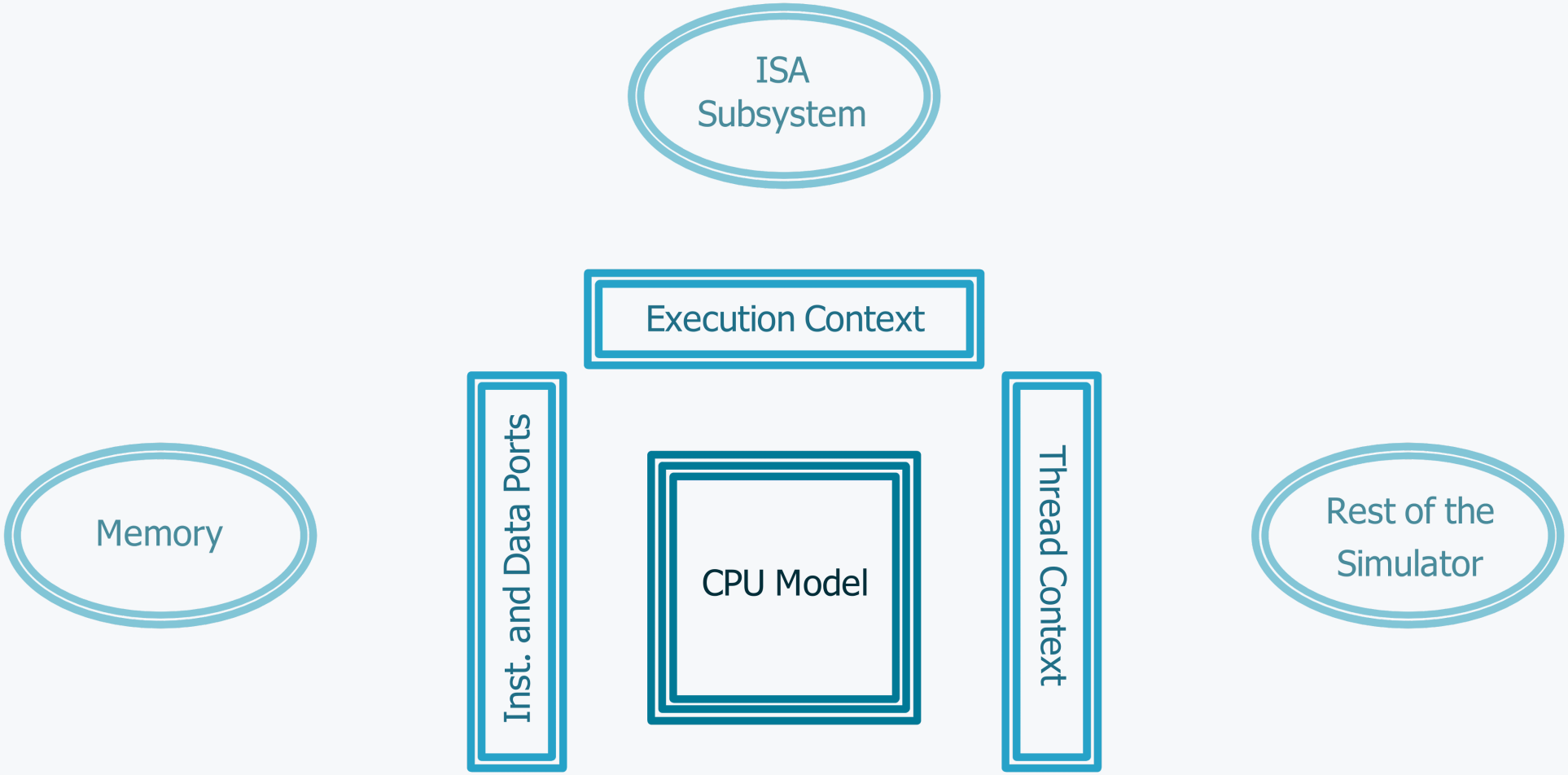
- Fast
- Some timing
- Caches, limited BP



DerivO3CPU and MinorCPU

- Slow
- Timing
- Caches, BP

Interaction of CPU model with other parts of gem5



Outline

- Learn about CPU models in gem5
  - AtomicSimpleCPU, TimingSimpleCPU, O3CPU, MinorCPU, KvmCPU
- **Using the CPU models**
  - Set up a simple system with two cache sizes and three CPU models
- Look at the gem5 generated statistics
  - To understand differences among CPU models
- Create a custom processor
  - Change parameters of a processor based on O3CPU

Exercise 1: Comparing CPU models

In this exercise, we will compare the performance of different CPU models in gem5.
You will compare the performance of the AtomicSimpleCPU, TimingSimpleCPU, MinorCPU, and O3CPU.
Use the "riscv-matrix-multiply-run" workload.

Questions:

1. Compare the total number of instructions simulated by each CPU model. What do you observe?
2. Compare the total simulated time for each CPU model. What do you observe?
3. Change the parameters of the cache hierarchy and compare the performance of the Atomic and Timing CPU models. What do you observe?

Step 1: Create the processor

In your script, use the `SimpleProcessor`.
See `SimpleProcessor` for more information.

Since you're going to be running the "riscv-matrix-multiply" workload, make sure you set up your processor to use the RISC-V ISA.

The workload is single threaded, so there's no need to set up multiple cores.

Choose the appropriate CPU type for the processor.

Step 1: Answer

```
processor = SimpleProcessor(
    cpu_type=CPUTypes.ATOMIC,
    isa=ISA.RISCV,
    num_cores=1
)
```

You can also use `CPUTypes.TIMING` for the TimingSimpleCPU, `CPUTypes.MINOR` for the MinorCPU, and `CPUTypes.O3` for the O3CPU.

Step 2: Create the board, set up the workload, and run the simulation

You can use the `SimpleBoard` component to create a simple system with a single processor.
See `SimpleBoard` for more information.

With the `SimpleBoard`, you can set the processor, memory (use `SingleChannelDDR4_2400`), and cache hierarchy (use `PrivateL1CacheHierarchy`).

Then, use `obtain_resource` to get the workload and run the simulation.
See `obtain_resource` for more information.
We'll talk more about this in the next section.

> You may want to use `gem5 --outdir=<outdir-name>` to separate the output directories for each simulation.

Step 2: Answer

```
board = SimpleBoard(
    processor=processor,
    memory=SingleChannelDDR4_2400(),
    cache_hierarchy=PrivateL1CacheHierarchy()
)
workload = obtain_resource("riscv-matrix-multiply-run")
board.set_workload(workload)
sim = Simulator(board=board)
sim.run()
```

Run this with Atomic, Timing, Minor, and O3 CPUs.

Step 3: Change the cache size and compare atomic and timing

Change the cache size in the `PrivateL1CacheHierarchy` component and run the simulation again.

Exercise 1: Write Answers for These Questions

1. Compare the total number of instructions simulated by each CPU model. What do you observe?
2. Compare the total simulated time for each CPU model. What do you observe?
3. Change the parameters of the cache hierarchy and compare the performance of the Atomic and Timing CPU models. What do you observe?

Outline

- Learn about CPU models in gem5
  - AtomicSimpleCPU, TimingSimpleCPU, O3CPU, MinorCPU, KvmCPU
- Using the CPU models
  - Set-up a simple system with two cache sizes and three CPU models
- **Look at the gem5 generated statistics**
  - To understand differences among CPU models
- Create a custom processor
  - Change parameters of a processor based on O3CPU

Statistics

Look at the Number of Operations

Assuming you called your output directories `atomic-normal-cache`, `atomic-small-cache`, `timing-normal-cache`, and `timing-small-cache`, you can look at the number of operations (instructions) simulated by each CPU model.

Run the following command.

```
grep "simOps" *cache/stats.txt
```

Here are the expected results. (Note: Some text is removed for readability.)

```
atomic-normal-cache/stats.txt:simOps                          33955020
atomic-small-cache/stats.txt:simOps                           33955020
timing-normal-cache/stats.txt:simOps                          33955020
timing-small-cache/stats.txt:simOps                           33955020
```

Ops vs Instructions

The ISAs in gem5 have some "microcoded" instructions that are broken down into multiple operations.
Thus, you will often see more "ops" than "instructions" in the statistics.

The x86 ISA has more complex instructions that are broken down into multiple operations.
In x86, you should expect to see almost double the number of ops compared to instructions.

It is the *ops* not the instructions that are executed by the CPU model.

Look at the Number of Execution Cycles

Run the following command.

```
grep -ri "cores0.*numCycles" *cache
```

Here are the expected results. (Note: Some text is removed for readability.)

```
atomic-normal-cache/stats.txt:board.processor.cores0.core.numCycles      38157549
atomic-small-cache/stats.txt:board.processor.cores0.core.numCycles       38157549
timing-normal-cache/stats.txt:board.processor.cores0.core.numCycles      62838389
timing-small-cache/stats.txt:board.processor.cores0.core.numCycles       96494522
```

Note that for Atomic CPU, the number of cycles is the **same** for a large cache *and* a small cache.

This is because Atomic CPU ignores memory access latency.

Extra Notes about gem5 Statistics

When you specify the out-directory for the stats file (when you use the flag `--outdir=<outdir-name>`), go to **`<outdir-name>/stats.txt`** to look at the entire statistics file.

For example, to look at the statistics file for the Atomic CPU with a small cache, go to **`atomic-small-cache/stats.txt`**.

In general, if you don't specify the out-directory, it will be **`m5out/stats.txt`**.

Other statistics to look at

- Simulated time (time simulated by gem5)
  - `simSeconds`
- Host time (time taken by gem5 to run your simulation)
  - `hostSeconds`

Outline

- Learn about CPU models in gem5
  - AtomicSimpleCPU, TimingSimpleCPU, O3CPU, MinorCPU, KvmCPU
- Using the CPU models
  - Set-up a simple system with two cache sizes and three CPU models
- Look at the gem5 generated statistics
  - To understand differences among CPU models
- **Create a custom processor**
  - Change parameters of a processor based on O3CPU

Exercise 2: Create a custom processor

Steps

1. Update class Big(O3CPU) and Little(O3CPU)
2. Run with Big processor
3. Run with Little processor
4. Compare statistics

We will be running the same workload (matrix-multiply) on **two custom processors**.

Questions

1. What is the IPC of the big and little processors?
2. What is the speedup of the big processor?
3. What is the bottleneck in the little processor?

Step 1: Create specialized models

In this step, you need to override the parameters of the O3CPU model to create two new classes: `BigO3` and `LittleO3`.

Remember, the O3CPU is a gem5 *SimObject* and we are going to wrap it to make it a standard library component.
This is a little more complicated for the processor since it is made up of many cores (and gem5 doesn't have a very well-defined core interface).

Create a new file called `my_processor.py` in the same directory as your script.
Add new classes, `BigO3` and `LittleO3`, to this file. They will inherit from the `RiscvO3CPU` class and override the parameters.

> Important: The CPU models are specialized for each ISA, so you need to make sure the ISA is included in your gem5 binary to have the RISC-V O3CPU.

Step 1: Continued

In your specialized classes, override the parameters of the O3CPU model to create the `BigO3` and `LittleO3` classes.

Specifically, change the `fetchWidth`, `decodeWidth`, `renameWidth`, `issueWidth`, `wbWidth`, `commitWidth` to either 2 or 8 (for little and big).

For the little processor, set the `numROBEntries`, `numIntRegs`, and `numFpRegs` parameters to 30, 40, and 40 respectively.

For the big processor, set the `numROBEntries`, `numIntRegs`, and `numFpRegs` parameters to 256, 512, and 512 respectively.

See `BaseO3CPU` for more information.

Step 1: Answer

```
class BigO3(RiscvO3CPU):
    def __init__(self):
        super().__init__()
        self.fetchWidth = 8
        self.decodeWidth = 8
        self.renameWidth = 8
        self.dispatchWidth = 8
        self.issueWidth = 8
        self.wbWidth = 8
        self.commitWidth = 8

        self.numROBEntries = 256
        self.numPhysIntRegs = 512
        self.numPhysFloatRegs = 512
```

```
class LittleO3(RiscvO3CPU):
    def __init__(self):
        super().__init__()
        self.fetchWidth = 2
        self.decodeWidth = 2
        self.renameWidth = 2
        self.dispatchWidth = 2
        self.issueWidth = 2
        self.wbWidth = 2
        self.commitWidth = 2

        self.numROBEntries = 30
        self.numPhysIntRegs = 40
        self.numPhysFloatRegs = 40
```

Step 2: Create stdlib core wrappers

In the same file, create two new classes: `BigCore` and `LittleCore`.
These classes will inherit from the `BaseCPUCore` class and override the `cpu` with your specialized O3CPU.

See `BaseCPUCore` for more information.

Step 2: Answer

```
class BigCore(BaseCPUCore):
    def __init__(self):
        core = BigO3()
        super().__init__(core, ISA.RISCV)

class LittleCore(BaseCPUCore):
    def __init__(self):
        core = LittleO3()
        super().__init__(core, ISA.RISCV)
```

Step 3: Create the processors

Now, add two new classes: `BigProcessor` and `LittleProcessor`.
These classes will inherit from the `BaseCPUProcessor` class and override the cores with your specialized O3CPU.
`BaseCPUProcessor` is a class that creates a stdlib processor with stdlib cores.

In the `BigProcessor` and `LittleProcessor` classes, you need to override the constructor to construct the cores.

See `BaseCPUProcessor` for more information.

Step 3: Answer

```
class BigProcessor(BaseCPUProcessor):
    def __init__(self):
        super().__init__(
            cores=[BigCore()]
        )

class LittleProcessor(BaseCPUProcessor):
    def __init__(self):
        super().__init__(
            cores=[LittleCore()]
        )
```

Step 4: Update the run script

Update the run script to use the custom processors.

Step 4: Answer

```
from my_processor import BigProcessor, LittleProcessor
if args.cpu_type == "Big":
    processor = BigProcessor()
elif args.cpu_type == "Little":
    processor = LittleProcessor()

board = SimpleBoard(
    clk_freq="3GHz",
    processor=processor,
    memory=SingleChannelDDR4_2400("1GiB"),
    cache_hierarchy=PrivateL1CacheHierarchy(
        l1d_size="32KiB", l1i_size="32KiB"
    ),
)
```

Exercise 2: Write Answers for These Questions

1. What is the IPC of the big and little processors?
2. What is the speedup of the big processor?
3. What is the bottleneck in the little processor?

Exercise 3: Exploring branch prediction effects

Now that you've created your `BigProcessor` and `LittleProcessor` let us look at the results of effective branch prediction on CPU models.

1. Modify `my_processor.py`'s `BigO3` class `__init__` function to include the following and re-run your run script.

```
self.branchPred = BiModeBP()
```

> Remember to import the branch predictor objects before using them:

```
from m5.objects.BranchPredictor import (
    BiModeBP,
    MultiperspectivePerceptronTAGE64KB,
)
```

Exercise 3 (cont): Exploring branch prediction effects

2. *Further* modify `my_processor.py`'s `BigO3` class `__init__` function to change the branch predictor again, and re-run your run script.

```
self.branchPred = MultiperspectivePerceptronTAGE64KB()
```

3. Compare your `stats.txt` results for your Big processors, focusing on `board.processor.cores.core.commit.branchMispredicts` counts.