

Modeling CPU cores in 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



gem5 CPU Models



Simple CPU



SimpleCPU

Atomic

Sequency 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.

MinorCPU



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



Let's use these CPU Models!



Material to use

Start by opening the following file.

<materials/02-Using-gem5/04-cores/cores.py>

Steps

1. Configure a simple system with Atomic CPU
2. Configure the same system with Timing CPU
3. Reduce the cache size
4. Change the CPU type back to Atomic

We will be running a workload called matrix-multiply on **different CPU types and cache sizes**.

Let's configure a simple system with Atomic CPU

[materials/02-Using-gem5/04-cores/cores.py](https://github.com/ARM-software/gem5/blob/master/materials/02-Using-gem5/04-cores/cores.py)

```
from gem5.components.boards.simple_board import SimpleBoard
from gem5.components.cachehierarchies.classic.private_l1_cache_hierarchy import PrivateL1CacheHierarchy
from gem5.components.memory.single_channel import SingleChannelDDR3_1600
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.isas import ISA
from gem5.resources.resource import obtain_resource
from gem5.simulate.simulator import Simulator

# A simple script to test with different CPU models
# We will run a simple application (matrix-multiply) with AtomicSimpleCPU and TimingSimpleCPU
# using two different cache sizes
...
```


Let's start with Atomic CPU

`cpu_type` in `cores.py` should already be set to Atomic.

```
# By default, use Atomic CPU
cpu_type = CPUTypes.ATOMIC

# Uncomment for steps 2 and 3
# cpu_type = CPUTypes.TIMING
```

Let's run it!

```
cd /workspaces/2024/materials/02-Using-gem5/04-cores
gem5 --outdir=atomic-normal-cache cores.py
```

Make sure the out directory is set to **atomic-normal-cache**.



Next, try Timing CPU

Change `cpu_type` in `cores.py` to Timing.

```
# By default, use Atomic CPU
# cpu_type = CPUTypes.ATOMIC

# Uncomment for steps 2 and 3
cpu_type = CPUTypes.TIMING
```

Let's run it!

```
gem5 --outdir=timing-normal-cache cores.py
```

Make sure the out directory is set to **timing-normal-cache**.



Now, try changing the Cache Size

Go to this line of code.

```
cache_hierarchy = PrivateL1CacheHierarchy(l1d_size="32KiB", l1i_size="32KiB")
```

Change `l1d_size` and `l1i_size` to 1KiB.

```
cache_hierarchy = PrivateL1CacheHierarchy(l1d_size="1KiB", l1i_size="1KiB")
```

Let's run it!

```
gem5 --outdir=timing-small-cache ./materials/02-Using-gem5/04-cores/cores.py
```

Make sure the out directory is set to **timing-small-cache**.

Now let's try a Small Cache with Atomic CPU

Set `cpu_type` in `cores.py` to Atomic.

```
# By default, use Atomic CPU
cpu_type = CPUTypes.ATOMIC

# Uncomment for steps 2 and 3
# cpu_type = CPUTypes.TIMING
```

Let's run it!

```
gem5 --outdir=atomic-small-cache cores.py
```

Make sure the out directory is set to **atomic-small-cache**.

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

Run the following command.

```
grep -ri "simOps" *cache
```

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

atomic-normal-cache/stats.txt:simOps	33954560
atomic-small-cache/stats.txt:simOps	33954560
timing-normal-cache/stats.txt:simOps	33954560
timing-small-cache/stats.txt:simOps	33954560

"Ops" may be different from "Instructions" because gem5 breaks instructions down into "micro-ops."

x86 is highly microcoded, all ISAs have some microcoded instructions in gem5.

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



Let's configure a custom processor!



Material to use

[materials/02-Using-gem5/04-cores/cores-complex.py](#)

[materials/02-Using-gem5/04-cores/components/processors.py](#)

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**.

Configuring two processors

We will make one fast processor (**Big**) and one slow processor (**Little**).

To do this, we will change **4** parameters in each processor.

- **width**
 - Width of fetch, decode, rename, issue, wb, and commit stages
- **rob_size**
 - The number of entries in the reorder buffer
- **num_int_regs**
 - The number of physical integer registers
- **num_fp_regs**
 - The number of physical vector/floating point registers

Configuring Big

Open the following file:

[materials/02-Using-gem5/04-cores/components/processors.py](#)

On the right, you'll see what `class Big` currently looks like.

Change the parameter values so that they are as follows:

- `width=10`
- `rob_size=40`
- `num_int_regs=50`
- `num_fp_regs=50`

```
class Big(O3CPU):  
    def __init__(self):  
        super().__init__(  
            width=0,  
            rob_size=0,  
            num_int_regs=0,  
            num_fp_regs=0,  
        )
```

Configuring Little

Now, on the right, you'll see what `class Little` currently looks like.

Change the parameter values so that they are as follows:

- `width=2`
- `rob_size=30`
- `num_int_regs=40`
- `num_fp_regs=40`

```
class Little(03CPU):  
    def __init__(self):  
        super().__init__(  
            width=0,  
            rob_size=0,  
            num_int_regs=0,  
            num_fp_regs=0,  
        )
```

Run with Big processor

We will be running the following file.

[materials/02-Using-gem5/04-cores/cores-complex.py](#)

First, we will run matrix-multiply with our **Big** processor.

Run with the following command:

```
gem5 --outdir=big-proc cores-complex.py -p big
```

Make sure the out directory is set to **big-proc**.

Run with Little processor

Next, we will run matrix-multiply with our `Little` processor.

Run with the following command:

```
gem5 --outdir=little-proc cores-complex.py -p little
```

Make sure the out directory is set to `little-proc`.

Comparing Big and Little processors

Run the following command.

```
grep -ri "simSeconds" *proc && grep -ri "numCycles" *proc
```

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

big-proc/stats.txt:simSeconds	0.028124
little-proc/stats.txt:simSeconds	0.036715
big-proc/stats.txt:board.processor.cores.core.numCycles	56247195
little-proc/stats.txt:board.processor.cores.core.numCycles	73430220

Our **Little** processor takes more time and more cycles than our **Big** processor.