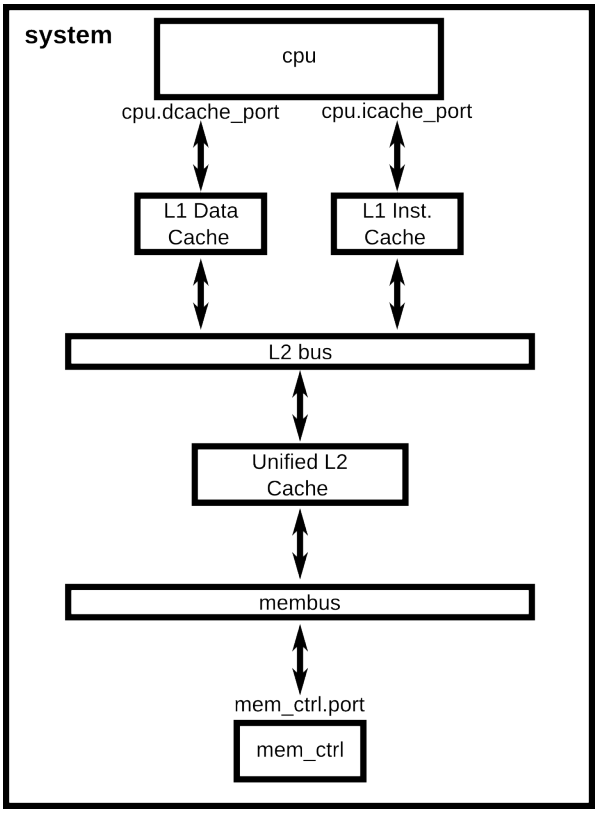
CSE 5600 Computer Systems Architecture I

Assignment 2, due Friday Oct. 9

In this lab assignment we will explore using the `gem5` simulator to look at the behavior of different CPU architectures and their issue width and how it affects their CPI and overall execution time.

1. In this exercise, we will be extending the system we created in the previous homework assignment,:



2. First, we are going to need to copy our old system from the assign1 folder. Issue this command from the `cse5600` folder to make a copy of the previous assignment which we can extend.

```
cp -r assign1 assign2
```

Now that we have our system copied, we are going to modify our cpu component. Open your `processor.py` file. First we need to modify the `make_processor` function. Right now it takes in one parameter (`isa`), and keeps the `cpu_type` constant. However, now we want to experiment with different CPU types, so we are going to need to change this constructor to build different CPU types.

First we want to add an import for a future config script we will make:

```
from config.branch_predictor import make_tournament_bp
```

Next, change the function signature to incorporate the new parameter:

```
def make_processor(isa, args):
```

Then we need to add a section to unpack this new parameter as soon as it is passed in. Immediately following the header, add:

```
    if args.cpu_model == "minor":
        cpu_type = CPUTypes.MINOR
    elif args.cpu_model == "o3":
        cpu_type = CPUTypes.O3
    else:
        raise ValueError(f"Unsupported CPU model: {args.cpu_model}")

    issue_width = args.issue_width
```

Once we have these values unpacked, we need to modify the SimpleProcessor construction to use the new cpu_type variable:

```
   processor = SimpleProcessor(
        cpu_type=cpu_type,
        isa=isa,
        num_cores=1,
    )
```

Now we need to wire in the `issue_width` variable we just made. In the previous assignment we set everything related to the core to 1, but now we want to sweep over different issue widths along with processor types. Where we previously set the core decode and execute parameters, replace the entire for loop with:

```
       for cpu in processor.get_cores():
        core = cpu.core

        core.branchPred.conditionalBranchPred = make_tournament_bp(
```

```
            bp_size=8192,
            bp_bits=2,
        )

        core.branchPred.btb.numEntries = 8192

        configure_width(
            core=core,
            cpu_model=cpu_type,
            width=issue_width,
        )
```

You will notice we also added a branch predictor, which we will add a factory for later. We also call a function called `configure_width`. We need to add this function in as well to handle the configuration easily for us based on passed in cpu type. Define this function as shown:

```
def configure_width(core, cpu_model, width):

    if cpu_model == CPUTypes.MINOR:
        core.decodeInputWidth = width
        core.executeInputWidth = width
        core.executeIssueLimit = width
        core.executeCommitLimit = width

    elif cpu_model == CPUTypes.O3:
        core.issueWidth = width

    else:
        raise ValueError(f"Unsupported CPU model: {cpu_model}")
```

Once you have this, save the file and close it.

3. Now we are going to add the branch predictor script. Create it in the config folder:

```
touch config/branch_predictor.py
```

Once you have this script created, open it up in the editor of your choice and add this into the file:

```
from m5.objects import TournamentBP

def make_tournament_bp(bp_size, bp_bits):
```

```
    bp = TournamentBP(
        localPredictorSize=bp_size,
        localCtrBits=bp_bits,
        globalPredictorSize=bp_size,
        globalCtrBits=bp_bits,
        choicePredictorSize=bp_size,
        choiceCtrBits=bp_bits,
    )

    return bp
```

This simple factory will make our branch predictor for our CPU which we wired-in in the previous step. Save this file and close it.

4. Now we just need to add some arguments to our `args.py` script to enable quick testing under our new system configuration. Open this script and add these options:

```
parser.add_argument(
    "--cpu-model",
    "--cpu_model",
    dest="cpu_model",
    default="o3",
    choices=["minor", "o3"],
)

parser.add_argument(
    "--issue-width",
    "--issue_width",
    dest="issue_width",
    default=1,
    choices=[1, 2],
    type=int,
)
```

Close and save this file too.

5. With our hardware components now described, we just need to make an entry-point script to call to construct and execute our simulations. Create a new file `O3_minor_single_double.py`. Open this file and add in all the needed imports just like we did with `x86_arm_riscv_compare.py`:

```
import atexit

import m5
from gem5.simulate.simulator import Simulator
```

```
from config.args import parse_args
from config.board import make_board
from config.cache import make_cache_hierarchy
from config.memory import make_memory
from config.processor import (
make_processor,
get_runtime_isa,
)
from config.workload import set_workload
from gem5.isas import ISA
from gem5.utils.requires import requires
requires(isa_required=ISA.RISCV)
```

Now we need to add in the section which takes in our arguments just like last time:

```
args = parse_args()
```

From these arguments we are going to construct our processor, cache hierarchy, memory and system board:

```
isa, isa_str = get_runtime_isa()

processor = make_processor(isa, args)
cache_hierarchy = make_cache_hierarchy()
memory = make_memory()

board = make_board(
    clk_freq="1.2GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)
```

Once that is done, we just need to add the workload and start the simulator. Add these final lines, and then save and close the file:

```
set_workload(board=board, args=args, isa_str="riscv")

simulator = Simulator(board=board)
simulator.run()

atexit.unregister(m5.stats.dump)
```

```
print(
    "Exiting @ tick {} because {}.".format(
        simulator.get_current_tick(),
        simulator.get_last_exit_event_cause(),
    )
)
```

6. Now that we have a working system, let's change some processor parameters and get some measurements. While we have the ability to vary the parameters of the cache and branch predictor, for this assignment, our interest is in the performance impact of the processor model and issue width.

   We can independently set the following things:

Parameter	Label	Possible Values
CPU model	`cpu-model`	`minor`, `o3`
issue width	`issue-width`	1, 2

   The `minor` model represents an in-order processor, while the `o3` model represents an out-of-order processor. The issue width controls how many instructions the processor may issue per cycle. All simulations in this assignment use the RISC-V ISA, a clock frequency of 1.2 GHz, and the same tournament branch predictor configuration. The cache hierarchy and memory configuration also remain fixed.

   For this assignment, execute every combination of the two CPU models and two issue widths for both applications, `daxpy` and `queens`. This requires four simulation runs per application, for a total of eight runs:

   - `minor` with an issue width of 1.
   - `minor` with an issue width of 2.
   - `o3` with an issue width of 1.
   - `o3` with an issue width of 2.

   Run each configuration with the RISC-V gem5 executable and pass the selected values using `--cpu-model` and `--issue-width`. For example:

```
$GEM5/build/RISCV/gem5.opt --outdir="$outdir" O3_minor_single_double.py
--prog="$app" --cpu-model="$cpu" --issue-width="$width"
```

   These simulations are short, but a simple Bash script is recommended to automate all eight runs and to give each output directory a distinct name based on the application, CPU model, and issue width.

   You do not need to include screenshots for every run; one example run is sufficient. Do, however, inspect the generated `config.ini` files to verify that each simulation used the intended CPU model and width. Record the CPI and total simulated execution time reported for each run.

7. For each application, plot CPI as a function of issue width. Use one data series for the `minor` processor and one for the `o3` processor. You may place both applications on one graph if all four series and their data points remain easy to distinguish, or use a separate graph for `daxpy` and `queens`. To improve readability, the y-axis may begin at a value greater than zero, but its scale should be clearly labeled.

   Describe in a paragraph or so what you can conclude from the data. Compare the in-order and out-of-order processor models and explain how increasing the issue width from 1 to 2 affects CPI. Does the wider issue width provide the same benefit for both processor models? Does it provide the same benefit for `daxpy` and `queens`? Are the results what you expected, and are there any surprises? Support your conclusions with the measured CPI values and, where useful, the corresponding execution times.