Computer Architectures	Delivery date: Monday 15/11
Laboratory 5	Expected delivery of lab_05.zip must include: - this document compiled possibly in pdf format.

Processor configuration and performance checking in gem5

1) Download from the course site the supporting material: lab5_material.zip
Unzip the file in your working directory (example `my_gem5Dir`) and obtain the following files:

1) `start.sh`
   a bash script that correctly sets the gem5 paths to execute the python script: `mygem5script.py`.

2) `mygem5script.py`
   a configurable script for gem5 that allows you to set different features to the simulated processor. In a few words, the script configures an Out-of-Order (O3) processor based on the *DerivO3CPU*, a superscalar processor, with a reduce number of features.
   The processor pipeline stages can be summarized as:

   - *Fetch stage*: instructions are fetched from the instruction cache. The `fetchWidth` parameter set the number of fetched instructions. This stage does branch prediction and branch target prediction.
   - *Decode stage*: This stage decode instructions and handles execution of unconditional branches. The `decodeWidth` parameter sets the maximum number of instructions processed per clock cycle.
   - *Rename stage*: parameters relevant for this stage are the entries in the re-order buffer and the instruction queue (a kind of shared reservation stations). Register operands of the instruction are renamed, updating a renaming map (stall may appear if not available entries). The maximum number of instructions processed per clock cycle is set by the `renameWidth` parameter.
   - *Dispatch/issue stage*: instructions whose renamed operands are available are dispatched to functional units. For loads, stores, they are dispatched to the Load/Store Queue (LSQ). The simulated processor has a single instruction queue from which all instructions issue. Ordinarily instructions are taken in-order from this queue. The maximum number of instructions processed per clock cycle is set by the `dispatchWidth` parameter.
   - *Execute stage*: the functional unit actually processes their instruction. Each functional can be configured with a different latency. Conditional branch mispredictions are identified here. The maximum number of instructions processed per clock cycle depends on the different functional units configured and their latencies.
   - *Write stage*: it sends the result of the instruction to the reorder buffet. The maximum number of instructions processed per clock cycle is set by the `wbWidth` parameter.
   - *Commit stage*: it processes the reorder buffer, freeing up reorder buffer entries. The maximum number of instructions processed per clock cycle is set by the `commitWidth` parameter.

   In the event of branch misprediction, trap, or other speculative execution event, "squashing" can occur at all stages of this pipeline. When a pending instruction is squashed, it is removed from the instruction queues, reorder buffers, requests to the instruction cache, etc.

2) Simulate the program `basicmath_large` (from MiBench) following the next steps. Remember to modify the program in order to **reduce the simulation time**. Please write here the changes that you have done in your program (`basicmath_large`):

a. Run the `start.sh` script for setting the gem5 paths

```
~/my_gem5Dir$ source start.sh
```

b. Simulate the program

```
~/my_gem5Dir$ /opt/gem5/build/ALPHA/gem5.opt mygem5script.py -c basicmath_large
```

Notice that the program output is automatically redirected to the file `m5out/program.out`.

Check the statistics (in `m5out`) file and collect the following parameters:

a) Number of instructions simulated
b) Number of CPU Clock Cycles
c) Clock Cycles per Instruction (CPI)
d) Number of instructions committed
e) Host time in seconds
f) Prediction ratio for Conditional Branches
   - Prediction ratio = Number of Incorrect Predicted Conditional Branches / Number of Predicted Conditional Branches
g) BTB hits.

Collect these parameters in Table 1 in the column *Basic configuration*.

3) Modify the processor configuration by doubling the parameters in the stages: Fetch, decode, rename, dispatch, execute, write and commit. **Do not change any value related to the branch predictors**. Simulate again the program `basicmath_large` and collect the statistics in the Table 1 in the column *X2 configuration*.

4) Modify one more time the processor configuration by doubling again the parameters in the stages: Fetch, decode, rename, dispatch, execute, write and commit. **Do not change any value related to the branch predictors**. Simulate again the program `basicmath_large` and collect the statistics in the Table 1 in the column *X4 configuration*.

5) Analyzing the results obtained in the previous experiments, modify the *Basic configuration* of the processor in order to improve the previous results. **Do not change any value related to the branch predictors**. Simulate again the program `basicmath_large` and collect the statistics in the Table 1 in the column *Custom configuration*.

TABLE1: `basicmath_large` program behavior on different CPU configurations

CPUs / Parameters	Basic configuration	X2 configuration	X4 configuration	Custom configuration
Ticks				
CPU clock domain				
Clock Cycles				

Instructions simulated				
CPI				
Committed instructions				
Host seconds				
Prediction ratio				
BTB hits				

Report the processor configuration values of your custom configuration.

TABLE2: CPU custom configuration Vs. the basic one

Parameter name	*Basic configuration* value	New value

2) Branch predictors comparison

The gem5 includes different branch predictors:

- LocalBP:
  Implements a local predictor that uses the PC to index into a table of counters. It is similar to a basic BHT.

- BiModeBP:
  The bi-mode predictor is a two-level branch predictor that has three separate history arrays: a taken array, a not-taken array, and a choice array. The taken/not-taken arrays are indexed by a hash of the PC and the global history. The choice array is indexed by the PC only. Because the taken/not-taken arrays use the same index, they must be the same size.
  The bi-mode branch predictor aims to eliminate the destructive aliasing that occurs when two branches of opposite biases share the same global history pattern. By separating the predictors into taken/not-taken arrays, and using the branch's PC to choose between the two, destructive aliasing is reduced.

- TournamentBP:
  Implements a tournament branch predictor, hopefully identical to the one used in the 21264. It has a local predictor, which uses a local history table to index into a table of counters, and a global predictor, which uses a global history to index into a table of counters. A choice predictor chooses between the two. Both the global history register and the selected local history are speculatively updated.

Starting from your Custom Configuration, enable one at a time, every one of the different branch predictors in the `mygem5script.py` section called: `BPU SELECTION`, and collect the resulting statistics for any configuration in the following table. Select one of the branch predictors and customize its values. Report the results in the last column of the next table.

TABLE3: `basicmath_large` program behavior on different CPU configurations

CPUs Parameters	Local predictor	Bimodal predictor	Tournament predictor	Custom configuration
Ticks				
CPU clock domain				
Clock Cycles				
Instructions simulated				
CPI				
Committed instructions				
Host seconds				
Prediction ratio				
BTB hits				

Report the branch prediction configuration of your custom configuration.

TABLE4: BPU custom configuration Vs. the basic one

Parameter name	*Basic configuration* value	New value