Computer Architectures 02LSEOV 02LSEOQ [AA-LZ]	Delivery date: November 22, 2020
Laboratory **5**	Expected delivery of lab_05.zip must include: - this document compiled in pdf format.

CPU Architect Engineer for one hour

1. Introduction

Set up the environment: download from the Portale della Didattica an updated version (gem5_env_2020_lab5.zip) of the gem5_env_2020 used in the previous session.

The environment is similar to the one used previously, but includes an additional benchmark (located in `benchmarks/basic_math`) and a python script `mygem5script.py`.

`mygem5script.py` is a configurable script for gem5 that allows you to set different features to the simulated processor. The script configures an Out-of-Order (O3) processor based on the *DerivO3CPU,* a superscalar processor, with a reduced 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*: send 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*: process 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.

Disclaimer: During this laboratory you are asked to play the role of a CPU architect engineer, in charge of designing a new processor. Therefore, you must rely on your own creativity and imagination.

2. 1st part: All the best money can buy

Initially, in the first part of the laboratory we will give you infinite resources. You are asked to design a new processor able to achieve the best performances money can buy. You can invest for improving the processor without any constraint.

The design of a new processor is guided by a set of benchmarks. They are used for evaluating whether the new architecture performs better compared to a previous one. For the evaluation, a set of statistics of interest are monitored. In our case, we will use only one benchmark and a limited set of statistics.

As done in the previous laboratory, do not forget to set up the environment with the `start.sh` script (or `start_vbox.sh`).

For this 1st part, we will use a new benchmark, the program `basicmath_small`. The program can be compiled with the MakeFile (command `make basicmath_small`).

Simulate the program with the following command

```
~/my_gem5Dir$ gem5_sim mygem5script.py  -c basicmath_small
```

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:

- `sim_ticks` (Number of ticks simulated)
- `sim_insts` (Number of instructions simulated)
- `system.cpu.numCycles` (Number of CPU Clock Cycles)
- `system.cpu.cpi` (Clock Cycles per Instruction)
- `system.cpu.committedInsts` (Number of instructions committed)
- `host_seconds` (Host time in seconds)
- Prediction ratio for Conditional Branches:
  `system.cpu.branchPred.condIncorrect` /
  `system.cpu.branchPred.condPredicted`
- `system.cpu.branchPred.BTBHits` (Number of BTB hits)

Collect these parameters in Table 1 in the column *Basic configuration*. This represents the starting point. From this configuration, you are asked to perform the so-called "Design Space Exploration". In this phase, different parameters are modified in order to find the best configuration. **Modify exclusively the parameters in the stages: Fetch, decode, rename, dispatch, execute, write and commit. Do not change any value related to the branch predictors.**

Experiment1: Modify the processor configuration by doubling the parameters in the stages: Fetch, decode, rename, dispatch, execute, write and commit. Simulate again the benchmark `basicmath_small` and collect the statistics in the Table 1 in the column *X2 configuration*.

Experiment2: Modify one more time the processor configuration by doubling again the parameters in the stages: Fetch, decode, rename, dispatch, execute, write and commit. Simulate again the benchmark `basicmath_small` and collect the statistics in the Table 1 in the column *X4 configuration*.

Now you should have an initial picture of the effects of the changes performed in Experiment 1 and 2. Analyze the results, and modify the basic processor configuration in order to improve the previous results as much as possible. **Again, modify exclusively the parameters in the stages: Fetch, decode, rename, dispatch, execute, write and commit. Do not change any value related to the branch predictors.**
For evaluating the effectiveness of your configuration use always the benchmark `basicmath_small` and collect the statistics in the Table 1 in the column *Custom configuration*.

TABLE1

CPUs Parameters	Basic configuration	X2 configuration	X4 configuration	Custom configuration
sim_ticks	10227552500	7098765000	6203797000	5968806500
sim_insts	8803313	8803313	8803313	8803313
system.cpu.numCycles	20455107	14197762	12407595	11937617
system.cpu.cpi	2.323569	1.612775	1.409423	1.356037
system.cpu.committedInsts	8803313	8803313	8803313	8803313
host_seconds	58.27	48.00	45.59	38.48
Prediction ratio	5.2615	5.9470	6.8781	6.9605
system.cpu.branchPred.BTBHits	807922	928205	1119327	1231735

Report also the processor configuration values of your custom configuration in Table 2 (increase the number of rows if required).

TABLE2: CPU custom configuration Vs. the basic one

Parameter name	Basic configuration value	New value
FetchWidth	2	8
FetchBufferSize	8	64
FetchQueueSize	8	32
DecodeWidth	2	8
NumIQEntries	4	16
NumROBEntries	4	256
RenameWidth	2	8
DispatchWidth	2	8
IssueWidth	2	8
LQEntries	8	32
SQEntries	8	32
WbWidth	2	8
CommitWidth	2	8
SquashWidth	2	8

When performing Design Space Exploration, only few parameters are changed at a time. Otherwise, the size of the problem would rapidly growth and become untreatable. Now that we have a stable

configuration (your *Custom Configuration* derived previously) it is time to consider also different units. Let's move to Branch predictors.
The gem5 includes different branch predictors:

- LocalBP:
  Implements a local predictor that uses the PC to index into a table of predictors. 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, each one of the different branch predictors in the `mygem5script.py` section called: `BPU SELECTION`. Collect the resulting statistics for each configuration in the Table 3. Select exclusively **one (don't ask which one you should select)** of the branch predictors and customize its values. Report the results in the last column labeled *BP Custom Configuration*.

TABLE3

CPUs Parameters	LocalBP	BiModeBP	TournamentBP	BP Custom configuration
sim_ticks	5968806500	5867690500	5912942000	5291432500
sim_insts	8803313	8803313	8803313	8803313
system.cpu.numCycles	11937617	11735385	11825888	10582869
system.cpu.cpi	1.356037	1.333065	1.343345	1.202146
system.cpu.committedInsts	8803313	8803313	8803313	8803313
host_seconds	38.48	35.78	35.86	34.47
Prediction ratio	6.9605	7.2048	7.1661	13.1185
system.cpu.branchPred.BTBHits	1231735	1153597	1199985	1336671

In the following, report the branch predictor selected for the improvements and the branch prediction configuration of your custom configuration (Table 4).

Branch Predictor Selected: LocalBP

TABLE4:BPU custom configuration Vs. the basic one

Parameter name	Basic configuration value	New value
LocalPredictorSize	32	128
BTBEntries	256	4096

3. 2nd part: The reality

Unfortunately, the life of an engineer is not that easy. You are always requested to make choices, based on your intuition and experience. In particular, in a real-case scenario, you won't have all these degrees of freedom, but your manager will always give you a limited amount of resources in terms of engineers, computers and time to complete the duty. Additionally, you will work on a specific portion of the processor – not the entire one.

Simulations of a real set of benchmarks might take weeks, and implementing a new architectural feature requires months of coding and debugging (not just changing the value of one variable in a python script).

To get a glimpse of this, let's focus on the Functional Units only (see `FUNCTIONAL UNIT DEFINITION` in `mygem5script.py`). For each Functional Unit, we have the *count* (i.e., how many of those units are present) and a set of operation it can be used for. For each operation, it is also defined the *latency*.

For the following tasks, **consider exclusively the fft benchmark of the previous laboratory**. Preliminarily, simulate the benchmark `fft` with the Basic Configuration (i.e., the initial one that you have in the `mygem5script.py)` in order to obtain the `stats.txt` file.

Task 1: You are asked to improve the functional units responsible for the floating-point operations, by *halving their latency*. Consider the following operations:

- FloatAdd
- FloatCmp
- FloatCvt
- FloatMult
- FloatDiv

In this case, changing the latency can be done instantly but when considering a real processor this can take months of research. Therefore, before moving into the implementation part, you must have some insights that you are not going to waste your time and your company money.
Due to a limited budget, you are asked to select only one operation among the one listed above as target for the improvements.
Which one would you select? For answering this question, you have to resort (**again!**) to the Amdahl's Law.

During the Lab 2, we have used the formula with a great level of detail and accuracy since we were able to see clock cycle per clock cycle the evolution of the pipeline. However, in practice industrial

architectural simulators do not have this level of detail. Additionally, the benchmarks are long and complex programs. Consequently, it is unfeasible to analyse them as we did in the Lab2.

In this lab, you have to use the statics generated by gem5 (in `stats.txt`) and some approximations in order to compute all the elements composing the formula. Specifically, the Fraction Enhanced should be computed as follows (in the example, it is used the FloatMult):

$$Fraction_{enhanced} = \frac{(\text{system.cpu.iq.FU\_type\_0::FloatMult} * LatencyFloatMult)}{\text{system.cpu.numCycles}}$$

For each operation, compute the Speedup Overall resorting to the Amdahl's Law and then with the simulator. Report the obtained values in Table 5.

TABLE5

Operation	Franction enhanced (%)	Speedup Overall by Amdahl	Speedup Overall by Simulation
FloatAdd	5.856	1.0301	1.0188
FloatCmp	0.588	1.0029	1.0017
FloatCvt	0.515	1.0026	1.0001
FloatMult	3.850	1.0196	1.0133
FloatDiv	0.0007	1.0000035	1.000022

As you can see, the Amdahl's Law is a valuable tool for designers since it represents a good approximation of what the speedup will be (providing an upper bound), without running simulations.

Task 2: Probably, you have also noticed that for each Functional Unit, you can also increase how many of them are present in the processor. Now you are asked to decide, starting from the *Base Configuration and considering the fft benchmark*, for the following operations whether is more advantageous to:

- double the unit that implement that operation, or;
- halve the latency of that operation.

The operations to be considered are in the following Table 6. In the column *Action to be Taken* report which of the two option you would consider (Double Unit or Halve Latencty). This task cannot be done with the Amdahl's Law solely, therefore you can resort to gem5 for deciding.

TABLE6

Operation	Action to be Taken
IntALU	Doubling
FloatAdd	Halving
FloatCmp	Halving
FloatCvt	Halving
FloatMult	Halving
FloatDiv	Doubling

For each operation, provide also a motivation for the action to be taken that you have chosen. **Please note that motivations like "the speedup overall is grater in this case …." Are not accepted. Use the statistics and reason on their meaning.**

Your Motivation: The Integer ALU can be only doubled because it already has a latency of 1 and it can't be less then this value. The FloatingDiv I think is better to double because it can't be pipelined so it's better to double it while other units can be pipelined so it's better to halve the latency in this way we can finish more instructions in the half of the time.

As for the previous tasks, the budget allows for one action only (i.e., improvement). Once you have identified for each operation the most appropriate action to be taken, you are now requested to decide which action is the most urgent one (i.e., the one that gives the best results). Also here show your motivations.

Your answer and motivation: Looking at the number of FU instructions issued, I think that the most urgent improvement to do is the one regarding the Multiplication unit.

Hint: For your reasoning, focus on the statistics: instructions simulated, type of FU issued, attempts to use FU when none available, CPU cycles required and operation latency.