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	*10227884500*	*7168576000*	*6275652000*	*6055537000*
sim_insts	*8803354*	*8803354*	*8803354*	*8803354*
system.cpu.numCycles	*20456012*	*14337391*	*12551305*	*12111075*
system.cpu.cpi	*2.323661*	*1.628628*	*1.425741*	*1.375734*
system.cpu.committedInsts	*8803354*	*8803354*	*8803354*	*8803354*
host_seconds	*39.14*	*32.89*	*31.25*	*36.36*
Prediction ratio	*203478 / 1070237 =* *0.190124*	*213687 / 1263108 =* *0.169176*	*227157 / 1557795 =* *0.145820*	*248099 / 1840418 =* *0.134805*
system.cpu.branchPred.BTBHits	*808307*	*925106*	*1114454*	*1311610*

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
the_cpu.fetchWidth	*2*	*8*
the_cpu.fetchBufferSize	*8*	*64*
the_cpu.fetchQueueSize	*8*	*64*
the_cpu.decodeWidth	*2*	*4*
the_cpu.numIQEntries	*4*	*32*
the_cpu.numROBEntries	*4*	*32*
the_cpu.renameWidth	*2*	*4*
the_cpu.dispatchWidth	*2*	*8*
the_cpu.issueWidth	*2*	*8*
the_cpu.LQEntries	*8*	*16*
the_cpu.SQEntries	*8*	*16*
the_cpu.wbWidth	*2*	*8*
the_cpu.commitWidth	*2*	*8*
the_cpu.squashWidth	*2*	*32*

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 configuration	BiModeBP configuration	TournamentBP configuration	CustomBP configuration
sim_ticks	*6055537000*	*5961124500*	*5960385500*	*5476122000*
sim_insts	*8803354*	*8803354*	*8803354*	*8803354*
system.cpu.numCycles	*12111075*	*11922250*	*11920772*	*10952245*
system.cpu.cpi	*1.375734*	*1.354285*	*1.354117*	*1.244099*
system.cpu.committedInsts	*8803354*	*8803354*	*8803354*	*8803354*
host_seconds	*36.36*	*35.89*	*35.96*	*29.68*
Prediction ratio	*248099 / 1840418 = 0.134805*	*229871 / 1817649 = 0.126466*	*235960 / 1836595 = 0.128476*	*124422 / 1569115 = 0.079294*
system.cpu.branchPred.BTBHits	*1311610*	*1266574*	*1305258*	*1348376*

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: *BiModeBP*

TABLE4: BPU custom configuration Vs. the basic one

Parameter name	Basic configuration value	New value
my_predictor.BTBEntries	*256*	*1024*

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	Speedup Overall by Amdahl *	Speedup Overall by Simulation **
FloatAdd	*1.030189*	*40768205 / 39996493 =* *1.019294*
FloatCmp	*1.002907*	*40768205 / 40651035 =* *1.002882*
FloatCvt	*1.002586*	*40768205 / 40710395 =* *1.001420*
FloatMult	*1.019701*	*40768205 / 40214230 =* *1.013775*
FloatDiv	*1.000003*	*40768205 / 40768097 =* *1.000002*

*****

$$fraction_{enhanced}(FloatAdd) = \frac{system.cpu.iq.FU\_type\_0::FloatAdd \times LatencyFloadAdd}{\text{system.cpu.numCycles}} = \frac{238946 \times 10}{40768205} = 0.058610$$

$$speedup_{overall}(FloatAdd) = \frac{1}{(1 - fraction_{enhanced}(FloatAdd)) + \frac{fraction_{enhanced}(FloatAdd)}{speedup_{enhanced}(FloatAdd)}} = \frac{1}{(1 - 0.058610) + \frac{0.058610}{2}} = 1.030189$$

$$fraction_{enhanced}(FloatCmp) = \frac{system.cpu.iq.FU\_type\_0::FloatCmp \times LatencyFloadCmp}{\text{system.cpu.numCycles}} = \frac{23645 \times 10}{40768205} = 0.005799$$

$$speedup_{overall}(FloatCmp) = \frac{1}{(1 - fraction_{enhanced}(FloatCmp)) + \frac{fraction_{enhanced}(FloatCmp)}{speedup_{enhanced}(FloatCmp)}} = \frac{1}{(1 - 0.005799) + \frac{0.005799}{2}} = 1.002907$$

$$fraction_{enhanced}(FloatCvt) = \frac{system.cpu.iq.FU\_type\_0::FloatCvt \times LatencyFloadCvt}{\text{system.cpu.numCycles}} = \frac{21034 \times 10}{40768205} = 0.005159$$

$$speedup_{overall}(FloatCvt) = \frac{1}{(1 - fraction_{enhanced}(FloatCvt)) + \frac{fraction_{enhanced}(FloatCvt)}{speedup_{enhanced}(FloatCvt)}} = \frac{1}{(1 - 0.005159) + \frac{0.005159}{2}} = 1.002586$$

$$fraction_{enhanced}(FloatMult) = \frac{system.cpu.iq.FU\_type\_0::FloatMult \times LatencyFloadMult}{\text{system.cpu.numCycles}} = \frac{112524 \times 14}{40768205} = 0.038641$$

$$speedup_{overall}(FloatMult) = \frac{1}{(1 - fraction_{enhanced}(FloatMult)) + \frac{fraction_{enhanced}(FloatMult)}{speedup_{enhanced}(FloatMult)}} = \frac{1}{(1 - 0.038641) + \frac{0.038641}{2}} = 1.019701$$

$$fraction_{enhanced}(FloatDiv) = \frac{system.cpu.iq.FU\_type\_0::FloatDiv \times LatencyFloadDiv}{\text{system.cpu.numCycles}} = \frac{12 \times 24}{40768205} = 0.000007$$

$$speedup_{overall}(FloatDiv) = \frac{1}{(1 - fraction_{enhanced}(FloatDiv)) + \frac{fraction_{enhanced}(FloatDiv)}{speedup_{enhanced}(FloatDiv)}} = \frac{1}{(1 - 0.000007) + \frac{0.000007}{2}} = 1.000003$$

******

$$speedup_{overall} = \frac{system.cpu.numCycles_{old}}{system.cpu.numCycles_{new}}$$

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 Latency). 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`	*Double Unit*
`FloatAdd`	*Halve Latency*
`FloatCmp`	*Halve Latency*
`FloatCvt`	*Halve Latency*
`FloatMult`	*Halve Latency*
`FloatDiv`	*Halve Latency*

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 benchmark `fft` with the Basic Configuration provides important statistics about the number of instructions issued by functional units (IntALU; FP_ALU which contains FloatAdd, FloatCmp and FloatCvt units; FP_MultDiv which contains FloatMult and FloatDiv units) and the number of times a reservation buffer for a functional unit was not available but could have been used:

Functional Unit	Instructions issued	Attempts when not available
`IntALU`	*15955482*	*1769019*
`FloatAdd`	*238946*	*6513*
`FloatCmp`	*23645*	*35*
`FloatCvt`	*21034*	*2138*
`FloatMult`	*112524*	*91*
`FloatDiv`	*12*	*0*

If a functional unit has a high number of attempts to use it when unavailable, then it is probably necessary to double the unit to issue more than one instruction in parallel. If not, it is better to halve the latency.

For the IntALU functional unit it is only possible to double the unit because the latency is already 1 clock cycle and it cannot be halved.

For the FP_ALU functional unit, which contains FloatMult and FloatDiv units, the best solution is to halve the latency because the number of attempts to use when the unit is not available is negligible compared to the instructions issued.

For the FP_MultDiv functional unit, which contains FloatAdd, FloatCmp and FloatCvt units, it is not possible to make a choice a priori because the number of failed attempts to use FloatAdd and FloatCvt units is not negligible (for the FloatAdd unit approximately 30% of instructions have to wait before they can be issued). Gem5 can help to calculate the speedup overall if doubling the unit or halving the latency:

$$speedup_{overall}(FP\_MultDiv \rightarrow DoubleUnit) = \frac{system.cpu.numCycles_{old}}{system.cpu.numCycles_{new}} = \frac{40768205}{40763289} = 1.000121$$

$$speedup_{overall}(FP\_MultDiv \rightarrow HalveLatency) = \frac{system.cpu.numCycles_{old}}{system.cpu.numCycles_{new}} = \frac{40768205}{39833145} = 1.023474$$

From the results it is easy to determine that the best solution for the FP_MultDiv functional unit is to halve the latency.

Therefore, the benchmark `fft` with the actions taken (table 6) produces the following statistics:

Functional Unit	Instructions issued	Attempts when not available
IntALU	*15948716*	*0*
FloatAdd	*238937*	*9868*
FloatCmp	*23645*	*2802*
FloatCvt	*21034*	*3777*
FloatMult	*112524*	*131*
FloatDiv	*12*	*0*

As can be seen from the table, doubling the IntALU functional unit avoids attempts to use it when unavailable. Instead, as a consequence of halving the latency of the other functional units (FP_ALU and FP_MultDiv), there is a slight increase in the number of failed attempts to use these units, which is inevitable because it is only possible to choose whether to halve the latency or double the functional unit.
In conclusion, with the actions taken (table 6), the number of clock cycles required to commit an instruction (CPI) has improved from 2.082736 to 1.974121.

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:

According to the previous statistics and motivations, to decide which action is the most urgent, it is important to focus on units that have issued the more instructions and/or have the most latency. So, there are three possible alternatives for the most appropriate action to be taken:

- *Double (from 1 to 2) the Int_ALU functional unit, which issue 15955482 instructions with a latency of 1 clock cycle per instruction*
- *Halve the latency (from 10 to 5) of the FloatAdd unit, which issue 238946 instructions with a latency of 10 clock cycles per instruction*
- *Halve the latency (from 14 to 7) of the FloatMult unit, which issue 112524 instructions with a latency of 14 clock cycles per instruction*

For the first solution, by doubling the unit (that has 1 clock cycle latency), only a portion of the instruction issued can actually benefit from the action. Assuming an upper bound, 1769019 clock cycles can be spared, which are all the attempts to use the unit when unavailable.

For the second solution, by halving the latency of the unit, 5 clock cycles can be spared foreach instruction issued. In total, with 238937 instructions issued by the unit, a sparing of 1194685 clock cycles can be achieved, which is an upper bound.

For the third solution, by halving the latency of the unit, 7 clock cycles can be spared foreach instruction issued. In total, with 112524 instructions issued by the unit, a sparing of 787668 clock cycles can be achieved, which is an upper bound.

According to these assumptions, the first solution may be the best. With gem5 it is possible to evaluate the speedup overall and confirm or not the hypothesis made:

$$speedup_{overall}(Int\_ALU \rightarrow DoubleUnit) = \frac{system.cpu.numCycles_{old}}{system.cpu.numCycles_{new}} = \frac{40768205}{40174567} = 1.014776$$

$$speedup_{overall}(FloatAdd \rightarrow HalveLatency) = \frac{system.cpu.numCycles_{old}}{system.cpu.numCycles_{new}} = \frac{40768205}{39996493} = 1.019294$$

$$speedup_{overall}(FloatMult \rightarrow HalveLatency) = \frac{system.cpu.numCycles_{old}}{system.cpu.numCycles_{new}} = \frac{40768205}{40214230} = 1.013775$$

The results are slightly different from the assumptions made because it cannot be possible to determine the stalls caused by data dependencies. So, the best solution is to halve the latency of the FloatAdd unit.
In conclusion, with the action taken (halve the latency of the FloatAdd unit), the number of clock cycles required to commit an instruction (CPI) has improved from 2.082736 to 2.043312.

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.