Computer Architectures **[AAA-GRA]** **Laboratory** **5**	Delivery date: 18th November 2022 Expected delivery of lab_05.zip must include: - this document compiled possibly in pdf format.

1) Processor configuration and performance checking in gem5

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 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 the 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 station). 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 and stores, they are dispatched to the Load/Store Queue (LSQ). The simulated processor has a single instruction queue from which all instructions are issued. 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 unit 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 a 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`):

```
  /* Now solve some random equations */
  for(a1=1;a1<10;a1+=1) {
    for(b1=10;b1>0;b1-=1) {
      for(c1=5;c1<15;c1+=1) {
           for(d1=-1;d1>-5;d1-=1) {
                SolveCubic(a1, b1, c1, d1, &solutions, x);
                printf("Solutions:");
                for(i=0;i<solutions;i++)
                  printf(" %f",x[i]);
                printf("\n");
           }
      }
    }
  }


  printf("********* INTEGER SQR ROOTS ***********\n");
  /* perform some integer square roots */
  for (i = 0; i < 100; i+=2)
    {
      usqrt(i, &q);
                        // remainder differs on some machines
     // printf("sqrt(%3d) = %2d, remainder = %2d\n",
     printf("sqrt(%3d) = %2d\n",
             i, q.sqrt);
    }
  printf("\n");
  for (l = 0x3fed0169L; l < 0x3fed4169L; l++)
    {
         usqrt(l, &q);
         //printf("\nsqrt(%lX) = %X, remainder = %X\n", l, q.sqrt, q.frac);
         printf("sqrt(%lX) = %X\n", l, q.sqrt);
    }


  printf("********* ANGLE CONVERSION ***********\n");
  /* convert some rads to degrees */
/*   for (X = 0.0; X <= 360.0; X += 1.0) */
  for (X = 0.0; X <= 360.0; X += .1)
    printf("%3.0f degrees = %.12f radians\n", X, deg2rad(X));
  puts("");
/*   for (X = 0.0; X <= (2 * PI + 1e-6); X += (PI / 180)) */
  for (X = 0.0; X <= (2 * PI + 1Sse-6); X += (PI / 57))
    printf("%.12f radians = %3.0f degrees\n", X, rad2deg(X));


  return 0;
}
```

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*.

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*.

TABLE1: `basicmath_large` program behavior on different CPU configurations

CPUs Parameters	Basic configuration	X2 configuration	X4 configuration
Ticks	119781026500	88530225000	82152632000
CPU clock domain	500	500	500
Clock Cycles	239562053	177060450	164305264
Instructions simulated	94729465	94729465	94729465
CPI	2,528907484	1,869116964	1,734468404
Committed instructions	94729465	96398207	96398207
Host seconds	285.33	240.24	242.63
Prediction ratio	0,196249771	0,171834925	0,142344476
BTB hits	8485036	9940196	12907188

4) Select one of the previous hardware configurations (Basic, X2, X4):

Selected hardware Configuration:	X2

Despite hardware enhancements for increasing the CPU performances, remember that also optimizing compilers for programs in high-level code exist. The aim of optimizing compilers is to minimize or maximize some attributes of an executable computer program (code size, performance, etc.). They are also aware of hardware enhancements in order to perform very accurate optimizations.

Compilers can be your best friend (or worst enemy!). More information you provide in your program better the optimized program will be.

a) Compile the program `basicmath_large` using the provided *Makefile* using the ALPHA compiler with different optimization levels (DO NOT CONFUSE WITH O3 PROCESSOR).

hint:

add a variable to the Makefile in order to use change the optimization level:

```
OPT="-O3"
```

and substitute all the `-O3` occurrences with the new variable as follows:

```
-O3 → $(OPT)
```

b) For visualize the enabled optimizations from the compiler perspective, you can run:

```
~/my_gem5Dir$  /opt/alphaev67-unknown-linux-gnu/bin/alphaev67-unknown-linux-gnu-gcc -c -Q -O2 --help=optimizers
```

By changing the "-O2" parameter with the desired one, you will find the enabled/disabled optimizations.

Here are some possible types of optimization

- https://en.wikipedia.org/wiki/Optimizing_compiler

- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

c) Simulate the program for different optimization levels and collect statistics (change OPT variable in the Makefile, O3 is the default, you need to change OPT accordingly to the values in parenthesis).

TABLE2: `basicmath_large` program behavior on the different compiler optimization level

Optimization Parameters	Selected hw configuration (-O3)	Opt lvl 0 (-O0)	Opt lvl 1 (-O1)	Opt lvl 2 (-O2)	Opt size (-Os)	Opt fast (-O3 -ffast-math)
Ticks	88530225000	94094023000	84742009500	88530225000	86523016000	66721094000
CPU clock domain	500	500	500	500	500	500
Clock Cycles	177060450	188188046	169484019	177060450	173046032	133442188
Instructions simulated	94729465	108230792	94715330	94729465	94757095	81500989
CPI	1,869116964	1,738766228	1,789404302	1,869116964	1,826206597	1,637307591
Committed instructions	94729465	108230792	94715330	94729465	94757095	81500989
Host seconds	242.13	275.92	238.68	242.27	242.79	211.17
Prediction ratio	0,182864855	0,175797348	0,170101439	0,171834925	0,166650229	0,173881613
BTB hits	9940196	9764732	9639881	9940196	9795431	9368472
Executable Size	905268	905764	905268	905268	905268	838789

1) 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 and default optimization level (O3), 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	88530225000	86292616000	86623427000	85241324000
CPU clock domain	500	500	500	500
Clock Cycles	177060450	172585232	173246854	170482648
Instructions simulated	94729465	94729465	94729465	94729465
CPI	1,869116964	1,821874872	1,828859204	1,799679202
Committed instructions	94729465	94729465	94729465	94729465
Host seconds	242.13	514.56	510.34	529.32
Prediction ratio	0,182864855	0,149614463	0,149221164	0,118820310
BTB hits	9940196	9332295	9561388	10570086

Report the branch prediction configuration of your custom configuration.

BPU Custom Configuration chosen: **Local Predictor**

TABLE4: BPU custom configuration Vs. the basic one

Parameter name	*Basic configuration* value	New value
Local Predictor Size	32	128
BTB Entries	256	1024