Programming Homework 3 for CA

Due Dec 10th

You should do this assignment on your own. No late assignments!

If you have any problems, please contact the teaching assistant in time.

In this assignment you will learn about pipelined CPUs and non-pipelined CPUs. You will simulate a given program with TimingSimpleCPU to understand the instruction mix of the program. Then, you will simulate the same program with an pipelined inorder CPU to understand how the latency and bandwidth of different parts of pipeline affect performance.

Step 1

The DAXPY loop (double precision aX + Y) is an oft used operation in programs that work with matrices and vectors. The following code implements DAXPY in C++11.

```
#include <cstdio>
#include <random>

int main()
{
  const int N = 1000;
  double X[N], Y[N], alpha = 0.5;
  std::random_device rd; std::mt19937 gen(rd());
  std::uniform_real_distribution<> dis(1, 2);
  for (int i = 0; i < N; ++i)
  {
    X[i] = dis(gen);
    Y[i] = dis(gen);
  }

  // Start of daxpy loop
  for (int i = 0; i < N; ++i)
  {
    Y[i] = alpha * X[i] + Y[i];
  }
  // End of daxpy loop

  double sum = 0;
  for (int i = 0; i < N; ++i)
  {
    sum += Y[i];
  }
  printf("%lf\n", sum);
```

```
    return 0;
}
```

Your first task is to compile this code statically and simulate it with gem5 using the TimingSimpleCPU. Compile the program with -O2 flag to avoid running into unimplemented x87 instructions while simulating with gem5. Grep for op_class in the file stats.txt, find instructions for different op classes and analyze them. (For example, which part of the code they correspond to.)

Step 2

As you can see from the assembly code, instructions that are not central to the actual task of the program (computing aX + Y) will also be simulated. This includes the instructions for generating the vectors X and Y, summing elements in Y and printing the sum. There are only about 10-15 lines for the actual DAXPY loop while the assembly code are about 350 lines.

Usually, one would like to look only at statistics for the portion of the code that is most important. To do so, typically programs are annotated so that the simulator, on reaching an annotated portion of the code, carries out functions like create a checkpoint, output and reset statistical variables. To do this, you should edit the C++ code from the first part to output and reset stats just before the start of the DAXPY loop and just after it. For this, include the file util/m5/m5op.h in the program. You will find this file in util/m5 directory of the gem5 repository. Use the function m5_dump_reset_stats() from this file in your program. This function outputs the statistical variables and then resets them. You can provide 0 as the value for the delay and the period arguments.

Go to the directory `util/m5` and scon M5ops with following command:

```
scons build/x86/out/m5
```

Then learn and link M5 to your C++ code above. You can write a Makefile to do this.

Now again simulate the program with the timing simple CPU. This time you should see three sets of statistics in the file stats.txt.
Report the analysis of the three parts of the program same as the step 1. Provide the fragment of the generated assembly code that starts with call to m5_dump_reset_stats() and ends m5_dump_reset_stats(), and has the main daxpy loop in between.

Step 3

Now, change the CPU model from TimingSimpleCPU to O3CPU or MinorCPU, and simulate again.

If you use O3CPU, do visualization for O3CPU with this link. Select a short clip that can show pipeline, visualize it and stated in the report.

If you use MinorCPU, Take a look at the file `src/cpu/minor/MinorCPU.py`. In the definition of `MinorFU`, the class for functional units, it define two quantities `opLat` and `issueLat` . From the comments provided in the file, understand how these two parameters are to be used. Also note the different functional units that are instantiated as defined in class `MinorDefaultFUPool` . Assume that the `issueLat` and `opLat` of the FloatSimdFU can vary from 1 to 6 cycles and that they always sum to 7 cycles. For each decrease in the opLat, we need to pay with a unit increase in issueLat. Which design of the FloatSimd functional unit would you prefer, or to put it another way, what do you think is better for `opLat` and `issueLat` to equal? Provide statistical evidence obtained through simulations of the annotated portion of the code.

If you want, you can do both, and visualize the MinorCPU, which you can refer to this.

What to Hand In

Turn in your assignment on Canvas

- All files below should be attached as a zip file. The submission name should contain the name and ID numbers of the student, for example: zhangsan_921104681912_HW3.zip.
- A file named daxpy.cpp which is used for testing. This file should also include the pseudo-instructions (m5_dump_reset_stats()), and also provide a file daxpy.s with the fragment of the generated assembly code as asked for in step 2.
- stats.txt and config.ini files for all the simulations. Please do difference marking.
- A report on your experience, all the question asked, and answer the question:
  - What is the pipeline structure of CPU you try(O3/Minor)?
  - What is the difference between an pipeline and a non-pipeline in this assignment?