# Computer Architecture
# Instructor: Gedare Bloom
# Project 2

You must do this assignment in a group of 2-3 students. You may not use code downloaded from the Internet without prior permission. Always watch the course website for updates on the assignments.

Read all instructions carefully. **Start early.**

### *Due Monday, November 22, 11:59 P.M.*

## Group Formation

Agree to form a group with 1 or 2 of your classmates, and arrange to join the same Project2_Groups in Canvas (People → Groups). Do this ASAP.

## Pipeline Performance (60%)

Objective: Gain experience with pipeline simulation.

This assignment is based on exercise 3.6 of CA:AQA 3rd edition and the gem5 101 Part III, but you don't need to refer to them to complete this work.

You will simulate a given program with simple timing cpu to understand the instruction mix of the program. Then, you will simulate the same program with a pipelined inorder CPU to understand how the latency and bandwidth of different parts of pipeline affect performance. You will also be exposed to pseudo-instructions that are used for carrying out functions required by the underlying experiment.

### *1 Instruction Mix*

The priority queue is an abstract data structure that may be implemented many different ways. One possible interface for a priority queue is:

```
/* Allocates and initializes a new pq */
pq* pq_create();

/* Adds value to pq based on numerical order of key */
void pq_push(pq *head, double key, void *value);

/* Returns value from pq having the minimum key */
void* pq_pop(pq *head);

/* Deallocates (frees) pq. Shallow destruction,
meaning nodes in the pq are not recursively freed. */
void pq_destroy();
```

Where `pq` is an opaque type that represents the priority queue. This interface is defined in the provided pq.h file. Your job is to create two different priority queue implementations using C programming and analyze them in this assignment.

The first implementation will be based on a linked list that uses the insertion sorting algorithm. You'll create this implementation in a new file: pq-linklist.c

The second implementation will be based on a min-heap implemented as a binary tree. You'll create this implementation in a new file: pq-minheap.c

The following source code is also provided to you to use for testing your implementations.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#include "pq.h"

int main(int argc, char *argv[])
{
  int i;
  const int n = 1000;
  double *v = malloc(sizeof(double)*n);
  pq *mypq;

  /* init */
  srand(time(NULL));
  mypq = pq_create();
  for (i = 0; i < n; i++) v[i] = drand48();

  /* begin sort */
  for (i = 0; i < n; i++) pq_push(mypq, v[i], (void*)v[i]);
  for (i = 0; i < n; i++) v[i] = pq_pop(mypq);
  /* end sort */

  for (i = 0; i < n; i++) printf("%g\n", v[i]);

  free(v);
  return 0;
}
```

Your first task is to compile this code and simulate it with gem5/stable using the timing simple cpu. Compile the program with `-O2` (that's a capital letter o) flag to enable optimizations yet avoid running into unimplemented x87 instructions while simulating with gem5. Run with input argument (n) of 10000. Report the breakup of instructions for different op classes by using grep for "op_class" in the file stats.txt. Repeat for each of your pq-* implementations.

## 2 Profiling the Region of Interest

Generate the assembly code for the program above by using the -S and -O2 options when compiling with GCC. As you can see from the assembly code, instructions that are not central to the actual task of the program (sorting integers) will also be simulated. This includes the instructions for generating the inputs, and printing the sorted output.

Usually while carrying out experiments for evaluating a design, 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.

You will edit the C code from the first part to output and reset stats just before the start of the sort and just after it. For this, include the file m5ops.h in the program. You can see this file in include/gem5 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.

To link with the m5ops library you will need to do the following. First you need to build the libm5 library in your gem5 repo:

```
$ cd gem5
$ scons -C util/m5 build/X86/out/m5
```

Then you will need to link libm5 with your main executable, for example:

```
$ gcc -static -I${GEM5} -o main main.c pq-minheap.c \
    ${GEM5}/util/m5/build/X86/out/libm5.a
```

Where GEM5 is a shell env variable pointing to the root of your checked-out gem5 repo.

Now again simulate the program with the timing simple CPU. This time you should see three sets of statistics in the file stats.txt. The first is from the first call to m5_dump_reset_stats(), which represents execution statistics prior to the sort loops. The second is from measuring the sort loops. The third is the rest of execution until gem5 terminates. Report the breakup of instructions among different op classes for the three parts of the program again using n=1000. Provide the fragment of the generated assembly code that starts with a call to m5_dump_reset_stats() and ends at m5_dump_reset_stats(), and has the main sort loops in between. Repeat for each of your pq-* implementations.

## 3 Pipeline Design Exploration

There are several different types of CPUs that gem5 supports: atomic, timing, out-of-order, inorder and kvm. Let's talk about the timing and the inorder cpus. The timing CPU (also known as SimpleTimingCPU) executes each arithmetic instruction in a single cycle, but requires multiple cycles for memory accesses. Also, it is not pipelined. So only a single instruction is being worked upon at any time. The inorder cpu (also known as Minor) executes instructions in a pipelined fashion. As I understand it has the following pipe stages: fetch1, fetch2, decode and execute.

The Minor CPU requires a cache, so let's add caches. Use the following cache configuration: 32 KiB direct mapped L1 instruction cache, 64 KiB direct mapped L1 data cache, and 4 MiB 8-way L2 unified cache. Run your annotated program with the Minor CPU with n=1000, for each of your pq-* implementations. What are the L1 cache miss rates, and is that surprising/unsurprising to you? Which pq-* implementation is the best one, and how much faster is it than the others? Repeat with n=10 and n=10000.

Take a look at the file MinorCPU.py. In the definition of **MinorFU**, the class for functional units, we 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 the 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? Provide statistical evidence obtained through simulations of the annotated portion of the code that is surrounded by the m5_* function calls.

I provide two git-diff files with the changes required to add a custom cpu.py file and to make se.py use the provided cpu.py with the MyMinorCPU model. After applying the patches, you must use se.py with --cpu-type=MinorCPU --caches options. It will enable --fpu_operation_latency and --fpu_issue_latency options. You still need to figure out how to modify the cpu.py and se.py files to control the Integer ALUs (IntFUs) issue and op latencies. You'll have to modify both of these files to complete the next section.
*Note that TimingSimpleCPU won't work anymore with the modified se.py because I hard-coded MyMinorCPU.*

## *4 Functional Unit Design Tradeoffs*

The Minor CPU has by default two integer functional units as defined in the file MinorCPU.py (ignore the Multiplication and the Division units). Assume our original Minor CPU design requires 2 cycles for integer functions and 4 cycles for floating point functions operation latencies (opLat). In our upcoming Minor CPU, we can halve either of these latencies: integer or floating point. Which one should we go for? Provide statistical evidence obtained through simulations. *Use an issueLat of 1 (the default) for integer and floating point FUs*. In se.py, you need to add the argument parsing for int oplat and issuelat. And you need to add the relevant IntFU definitions in the derived MyMinorCPU class in cpu.py and instantiate them in the FUPool.

## *5 Compiler Optimization*

The compiler can have a significant impact on program performance. Often, an effective compiler optimization to reduce loop overhead is to use loop unrolling. Enable loop unrolling by adding the -funroll-loops option to the gcc command line. Re-compile your program with unrolled loops, using n = 1000. By examining the disassembled program, can you identify how many times the loops in main.c are unrolled? Did loops in any of your pq-* implementations get unrolled?

Repeat 3-4 with loop unrolling. Is loop unrolling good/bad/neutral for the performance of this implementation of the sort loop? Do any of your conclusions or design choices change due to unrolling?

## Report (40%)

Write and submit a file named report.pdf containing a short report with your observations and conclusions from the experiment. This report should contain:

- Your team members' names;
- The roles, responsibilities, and contributions of each team member;
- Answers to questions asked above, along with (descriptive statistics) statistical evidence to support your claims, in graph or table form;
- Anything else you would like to say.

## Submission Instructions (read carefully)

1. A file named main.c which is used for testing, and the pq-* implementation files: pq-linklist.c and pq-minheap.c. The main.c file should also include the pseudo-instructions (m5_dump_reset_stats()) as asked in part 2.2. Also provide a file main.S with the fragment of the generated assembly code as asked for in part 2, and another main_unrolled.S with the same fragment when loops are unrolled in part 5. **Delete the assembly code outside of the loop body.**
2. stats.txt and config.ini files for all the simulations you conducted.
3. The report.pdf file.

Make a tar and gzip file with your group members' UCCS usernames (email address without the @uccs.edu part) as the name.
– tar -zcvf ${USERNAME1}-${USERNAME2}-${USERNAME3}-project2.tgz ${USERNAME1}-${USERNAME2}-${USERNAME3}-project2/
Where all your files are in the ${USERNAME1}-${USERNAME2}-${USERNAME3} project2 directory. For example, I would replace ${USERNAME1} above with gbloom, and ${USERNAME2}/${USERNAME3} with the second/third members of my team. **Do not include compiled output in your submission!** Upload the tgz file to Project 2 on Canvas.