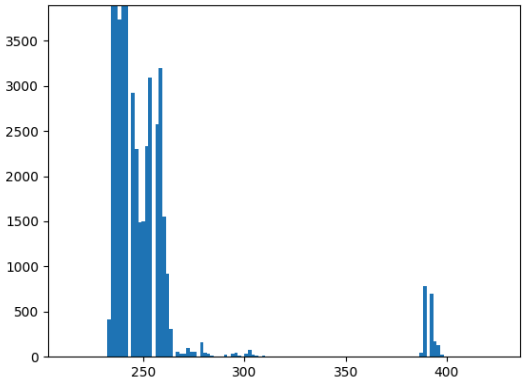
[IMPORTANT!] You can find the skeleton of the assignment in the `lab1.zip` archive on canvas. We ask you to use the skeleton and the Makefile we provide since we use them for grading. If the code does not compile with the given Makefile you will lose points.

[IMPORTANT!] The entire assignment must be written in C. In particular, assembly and inline assembly may not be used. Use the C wrappers to the assembly instructions provided in `asm.h` instead. These suffice to complete the assignment (you do not need to use them all). Note: you may not add additional inline assembly to `asm.h`. Only use what is provided.

Lab 1 - Manual

DRAMA[1] is a known side-channel attack used to leak information about the geometry of memory addresses in DRAM. The goal of this assignment is to partially reverse engineer the mapping between physical addresses and DRAM addresses. As we saw in the Memory Subsystem lecture, data inside DRAM is stored inside an 2d-array of cells known as a **bank**. A bank is composed of multiple rows. When you want to read data from these rows, you need to bring the content inside a structure known as **Row Buffer** that acts as a “cache”. Accessing two separate rows in the same bank causes a slow down in the memory accesses due to a conflict in the row buffer. As you can see in the plot below you can easily detect these conflicts when measuring time from software.



This lab is divided in three tasks:

1. In task 1 you will learn to detect these bank conflicts
2. In task 2 you will recover the physical address bits used for bank addressing
3. In task 3 (Bonus) you perform automatic conflict clustering to combine task 1 and 2

Deliverable:

- Run `make deliverable` to export a zip of the current directory containing:
  - The source code of your program implemented in `main.c`
  - A histogram like the one above for task 1
  - (optional) a README for any notes you want to submit

Task 1: Bank Conflict (6 points)

The goal of this part is to manage to plot a histogram like the one shown above.
You can simply mmap a large chunk of memory (e.g., 1GB) and generate a whole bunch of random `probe` addresses within this allocation (use the provided code stub). Then, for each of these random addresses you will measure the access time when accessed together with a `base` address (always the same). Plotting the frequency of the access times should give you a histogram like the one above. You can use the `plotter.py` script for that, cf. `make histogram`.
You will need this plot to identify a cut-off time to identify the addresses landing on the same bank (e.g., 350 cycles in the plot above).

Hints

Things can go wrong on multiple levels of the compute stack. Make sure you carefully think through what happens (vs what you want to happen) on every level. For example:

- *OS*: does demand-paging and/or (zero-)page deduplication impact the attack?
- *Compiler*: does the compiler really produce the binary code you expect?
- *CPU ISA*: how do you measure time accurately? Does the CPU's memory ordering model complicate things? Cf. `asm.h` for clues.
- *Microarchitecture*: how do you trigger memory accesses to DRAM (instead of the cache)? Does out-of-order execution complicate precise timing? And how about the prefetcher?

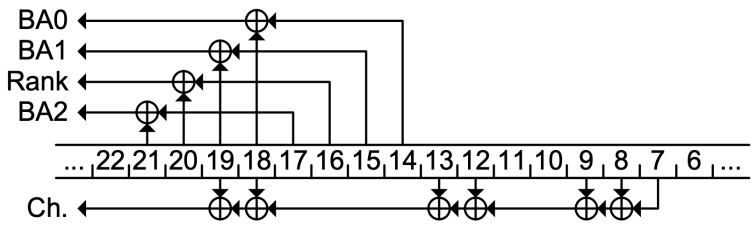
Even after taking the appropriate actions to overcome any potential problems listed above, you will not get a perfectly noise-free signal. Perform multiple measurements, look at the characteristics of your signal, and aggregate the results in a meaningful way. Maybe you can do better than just take the mean?

Different systems may have different cut-off frequencies due to different DRAM timings.

The number of addresses generating a bank conflict (i.e., right cluster on the histogram) depends on the memory configuration. For instance on a 1-channel, 1-DIMM, 2-rank, 16-bank configuration the number of conflicts should be around 1/(1 x 1 x 2 x 16) = 1/32 of the total addresses.

Task 2: Bank Bits recovery (4 pt)

In the previous task you managed to recover a cut-off time to detect addresses generating bank conflicts. In this second part we will try to recover the (physical) address bits used for the DRAM bank addressing functions. These functions are simply XORs of different bits of the physical address (Figure from the DRAMA paper [1]).



Intel Haswell (desktop system) – DDR3

In the example above you can see the functions for an Intel Haswell system. While in the figure you can distinguish between ranks, channels and banks address functions, from software you actually cannot detect the difference since the only side channel you have are the bank conflicts. For this reason you can consider the 1x2x8 configuration as a 16-banks configuration.
For this assignment we don't care about recovering the XOR-ing functions but only the bits used in them (e.g. bits 7-9,12-21 in the figure above). Since you do not have access to the physical address from userspace we remove one level of indirection (virtual→physical) by already mmap-ping a 1GB HugePages. In a 1GB HugePage the last 30 bits of the virtual and physical address are the same allowing you to identify any function using bits <=30.

You can now pick two addresses mapping to two different rows in the same bank (i.e.,bank conflicts) and flip bits on one of them to figure out when the address maps to a different bank.
The code skeleton we provide accepts the cut-off time as an argument of the binary.

Hints:

- You can identify two conflicting addresses by remembering from the lecture which address bits are used **ONLY** for the row index

Task 3 (Bonus): Automatic cut-off detection (1pt)

In this task we combine Task 1 and 2. The goal is to implement a function that automatically detects the cutoff value on the measurements you extracted in Task 1 and identify the bank hits and conflicts clusters automatically. After finding these two clusters you can identify a cutoff value and simply pass that to the `find_bank_bits` function.

References

[1] Pessl, P, et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." *arXiv* (2015): arXiv-1511.