Problem 1:

In this problem, we ran three separate simulations varying the fetch, decode, issue, and commit widths. When we doubled each of these values from 2 to 4, we got significant improvement in the IPC, 1.28 to 2.09 in mcf and 1.20 to 1.78 in equake. When we doubled each of the values again to 8, we saw only marginal increases in the IPC, 2.09 to 2.21 in mcf and 1.78 to 1.94 in equake. For each of the simulations there were a constant number of functional units.

For comparison purposes, we are assuming that the simulator default case (all values 4) produces the baseline IPC. When we switch the parameters to 2, the IPC is bound by the widths of the front end (fetch, decode, and issue widths). Basically, this means that portions of the processor are underutilized and that the code allows more parallelism than can be exploited by the limited front-end width. When we switched the values from 4 to 8, we saw only limited improvement in the IPC. This was caused by instructions being unable to issue due to the ruu or the lsq being full. In the third simulation, this happened a significant portion of the time. Basically, the front-end of the machine was bringing in instructions faster than the pipeline is capable of executing and committing them.

Problem 2:

For each of these simulations the fetch, decode, issue, and commit widths was 8. We assume that the simulation from Problem 1iii produces our baseline IPC. In this situation, we were bound by the sizes of the ruu and lsq. For the first simulation in Problem 2, we doubled the number of functional units. We initially thought that this would greatly improve the IPC, but it lead to a minor decrease in the IPC, from 2.21 to 2.20 in mcf and equake remained the same. When we increased the number of functional units, we improved the instances where issue was blocked due to unavailable functional units. This was only a small percentage of cases and their improvement lead to slightly greater contention for the ruu and lsq. In summary, the simulations were initially bound by the lsq and ruu sizes. We made this worse by doubling the functional units.

In Problem 2ii, we combated this contention by doubling the size of the ruu and lsq. This greatly improved the IPC, from 2.21 to 2.72 in mcf and 1.94 to 2.28 in equake. In this case, full lsq and ruu conditions block issue much less of the time. Therefore, the additional functional units that were added in Problem 2i could be utilized.

* All noted values are in the table on the page that follows the report.
Problem 3:

We assume that the configuration from Problem 2ii is the baseline IPC. In Problem 3i, we cut the issue and commit widths in half. This causes a significant drop in the IPC, from 2.72 to 2.37 in mcf and 2.28 to 2.04 in equake. The IPC drops for two reasons: first (and less significant) is the fact that we cannot issue as many instructions to the functional units thus leaving some of them underutilized. The second and most significant reason for the decrease in IPC is the inability to free entries in the ruu. This is caused by the limited commit width. This is reflected in the huge increase in the ruu full percentages that cause issue to be stalled. Both of these factors contribute to the increase in the IFQ occupancy.

In Problem 3ii, we again started from the configuration in Problem 2ii and cut the decode width from 8 to 4. This also caused a decrease in the IPC, from 2.72 to 2.52 in mcf and from 2.28 to 2.03 in equake. In this case the front-end is not capable of decoding enough instructions to keep the pipeline full. This is reflected in the occupancy percentages of both the ruu and lsq. Essentially, the number of instructions that can be decoded limits the IPC.