CS/ECE 752
Fall 2024
HW 2 Solution & Rubric
Professor Matt Sinclair

Report:

1. Report the breakdown of instructions for different op classes and provide a brief analysis of the breakdown.

The key thing to identify here is that the instruction breakdown in the startup and teardown phases is significantly different from the instruction breakdown in the region of interest (ROI). For example, the initial phase before the ROI includes random number generation initialization and usage, allocating arrays, and initializing arrays. Moreover, there are often a significant number of instructions run just to setup the program. Similarly, at the end of the program (after the ROI) there are a significant number of instructions devoted to summing, printing, and freeing. This affects our instruction breakdown and – if not using the ROI labels – can influence what sort of instructions we view as important in our program.

However, in this problem we have not put the ROI markers in. Thus, we will see the instruction breakdown weighted towards the operations in the startup and teardown phases. Specifically, IntALU (> 50%), MemRead (> 10%), and MemWrite (> 5%) operations are likely to dominate (which makes sense because the startup/teardown phases have a lot of these operations), and you should see that Float and SIMD operations are < 5% each (despite them being the part we care the most about!).

2. After adding the m5_dump_reset_stats flags to your program, report the instruction breakdown for different op classes in each of the 3 parts of your stats file and provide a brief analysis of the breakdown.

(Note: this answer focuses on the analysis of the instruction breakdown, but your assembly snippet should also be included)

As mentioned above, now that we add the ROI markers in, now we should see very different instruction breakdowns. The setup and teardown phases should still show that IntALU, MemRead, and MemWrite instructions dominate (with similar percentages to before). This makes sense because they are spending most of their time initializing arrays (e.g., reading and writing those arrays), as well as performing ALU operations. However, the ROI region – the region we care the most about – shows that Float and SIMD instructions are a much larger percentage than they were in Problem 1! Although IntALU operations still consume a large percentage (~50%), now FloatMemRead, FloatMemWrite, SimdFloatAdd, and SimdFloatMult should all be a bigger portion of the instructions each (especially FloatMemRead since we are reading two operands per instruction). Moreover, you should be able to see that the number of these operations directly mirrors the number of loop iterations you are executing – e.g., 4096 SimdFloatAdd operations. This makes sense, because our ROI is executing the loop where the math operations occur, and we know exactly how many adds and multiplies are occurring in this loop. You may also notice that, when comparing the total instructions for each of the 3 stats subsets, that the ROI accounts for ~1% of the overall instructions, despite being the part we care the most about! This highlights the importance of marking the part(s) of the program we care about – because we

could draw very different conclusions if we focused on the entire program vs. the most important part we are trying to optimize for.

3. Which design of the FloatSimd functional unit would you prefer? Provide statistical evidence obtained through simulations of the annotated portion of the code.

In general, you should see a trend towards preferring functional unit (FU) designs with lower issue latency, even if it comes at the expense of operation latency (i.e., opLat = 6, issueLat = 1). This is because MinorCPU is an in-order processor and (if you examine the documentation or files like func_unit.cc) issue latency stalls the whole pipeline (which makes sense, because the processor is in-order – anything after the current instruction must also stall to maintain von Neumann semantics!). In comparison, operation latencies do not necessarily stall the pipeline because the processor has multiple FUs that can be used simultaneously during the execution phase. Thus, it makes sense that the MinorCPU processor is more sensitive to issue latency.

(Note: if you analyzed the entire program instead of just the ROI, you likely found opLat = 4, issueLat = 3 is the best, because different types of instructions dominate the setup and teardown phases – your analysis should have explained why this is the case.)

4. Should we prefer halving the operation latency for the Integer functions or floating point functions? Provide statistical evidence obtained through simulation.

The baseline MinorCPU configuration has two integer FUs (2 cycles operation latency) and 1 FP FU (4 cycles operation latency). Moreover, each of the FU types also has 1 cycle issue latency in the default MinorCPU code. When it comes to halving the operation latency of either, the simulation results suggest that halving the operation latency for the integer operations vs. floating point operations provides similar performance benefits. However, halving the FP unit provides slightly better performance improvement. It makes sense that the results are similar, because there are (roughly) twice as many integer ALU operations in our loop as there are FP operations. However, the FP operations must all go to a single unit, whereas the ALU operations have two. Thus, it makes sense that the single FP unit and its longer operation latency would present more of a bottleneck.