LSU EE 4720 **Homework 5** Solution **Due: 3 December 2002**

To answer the questions below you need to use the PSE dataset viewer program. PSE (pronounced see) runs on Solaris and Linux; you can use the computer accounts distributed in class to run it, a Linux distribution may also be provided for running it on other systems.

Procedures for setting up the class account and using PSE are at `http://www.ece.lsu.edu/ee4720/proc.html`; preliminary documentation for PSE is at `http://www.ece.lsu.edu/ee4720/pse.pdf`.

Problem 1: Near the beginning of the semester the performance of a program to compute π was evaluated with and without optimization. It's back, down below.

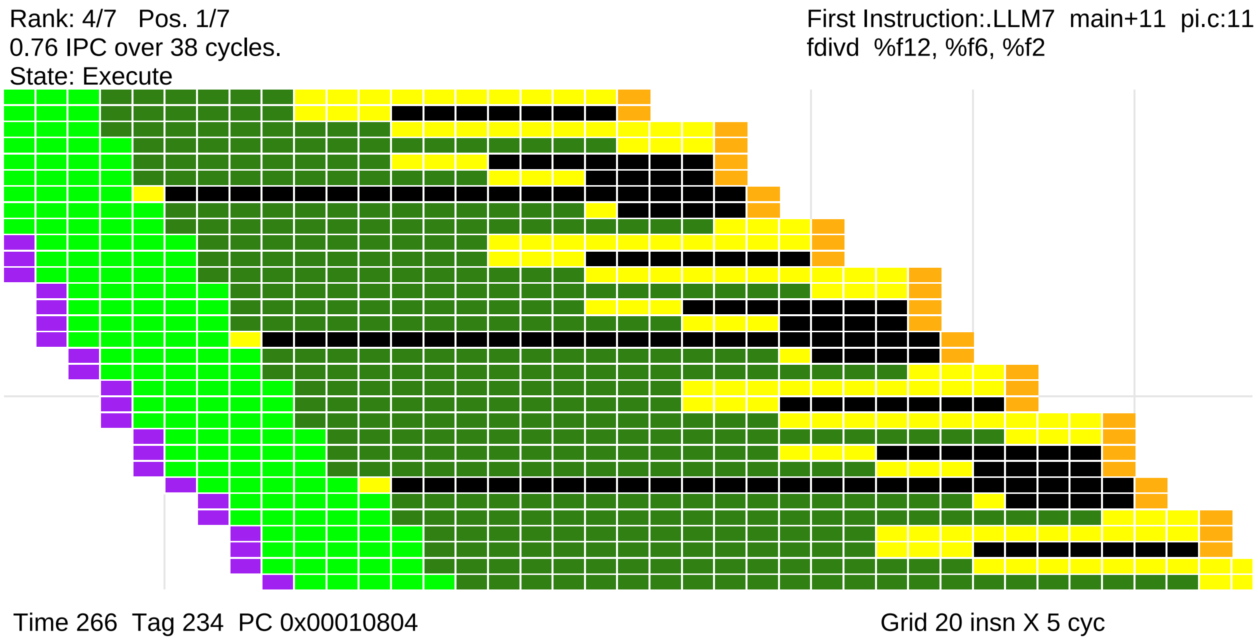
Follow instructions referred to above to view the execution of the optimized and unoptimized versions of the pi program running on a simulated 4-way dynamically scheduled superscalar machine with a 48-instruction reorder buffer. The datasets to use are `pi_opt.ds` and `pi_noopt.ds`.

(*a*) Based on the pipeline execution diagram compute the CPI of the main loop for a large number of iterations in the optimized version. Do not use the IPC displayed by PSE, instead base it on the PED. In your answer describe how the CPI was determined.

To find the precise CPI first find a repeating pattern. Fortunately, once the branch predictor warms up and the ROB fills each iteration is identical so a unit of the repeating pattern is one iteration long. One such iteration (not the first) starts at cycle (time) `339`, the next starts at `345`, for a time of `6` cycles. There are `9` instructions (including the `nop`), so the CPI is $\frac{6}{9} = \frac{2}{3}$.

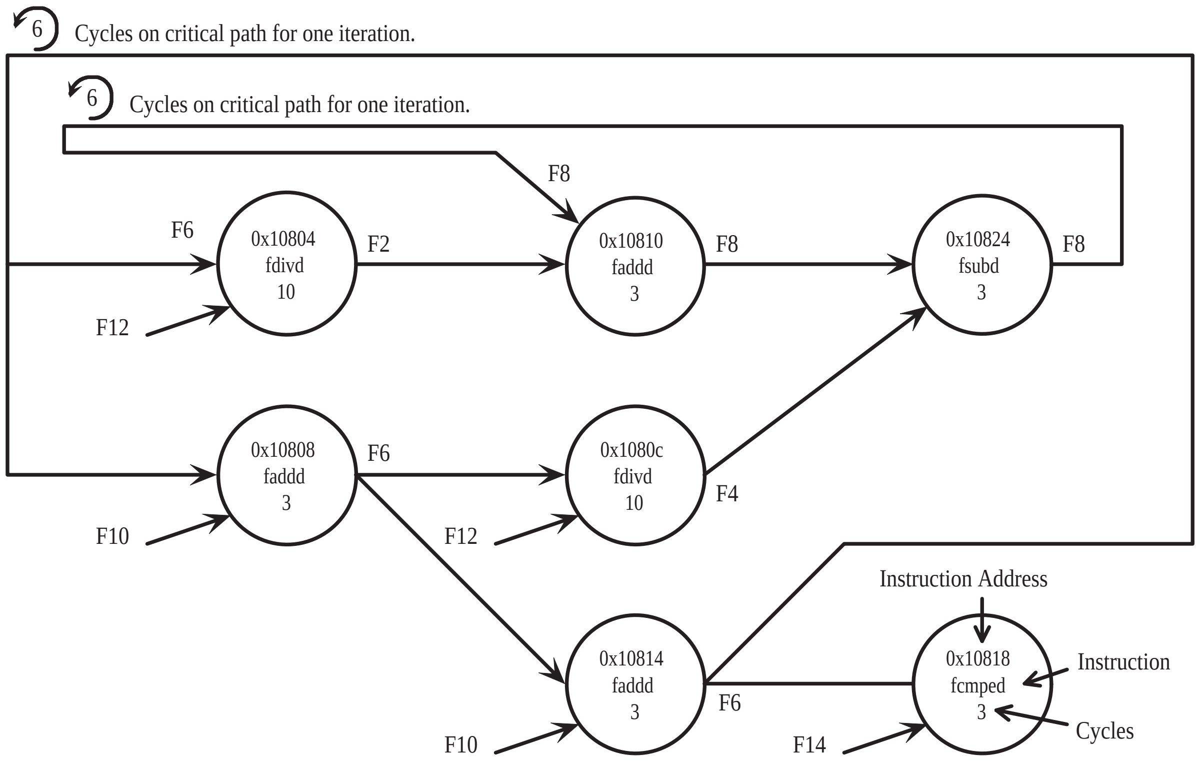
(*b*) Consider first the optimized version of the program. Would it run faster with a larger reorder buffer? Would it run faster on an 8-way superscalar machine? How else might the processor be modified to improve performance? Explain each answer.

An important feature to notice is that, except for `nop`, instructions wait many cycles before executing. All of the waiting instructions are waiting for operands and so execution time is limited by the critical path through the code. (No instruction in the loop waits for a functional unit, there are enough for this loop.)



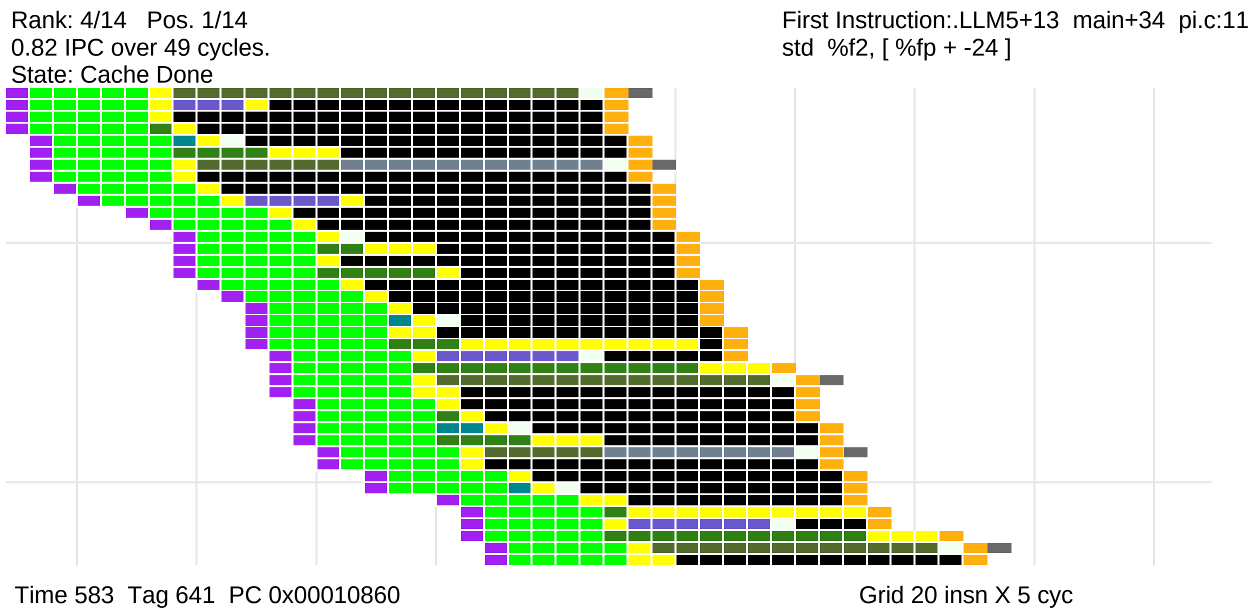
With a larger ROB there will be more instructions in flight, but all these "new" instructions will do is wait. Similarly, an 8-way machine will fetch instructions twice as fast and provide additional functional units, but that won't change the critical path. All an 8-way processor will do for this loop is fill the ROB more quickly.

The problem can be more precisely solved by constructing a dataflow graph (DFG) and finding the critical path. The dataflow graph for this loop appears below:



The loop critical path is two floating point adds, taking three cycles each. Actually two paths are tied for the critical path award: one path is 0x10810 $\rightarrow$0x10824 and the other is 0x10808 $\rightarrow$0x10814. Since the add instructions wait only for operands, performance can be improved if the floating-point adder takes fewer cycles (say, two), based on what was covered in the class. A more exotic solution would be to have a floating-point functional unit that can perform three-source-operand instructions and a processor that could recognize pairs of instructions that could be replaced by three-source-operand instructions. (The question asked about processor modifications, not compiler or ISA modifications, so one could not just re-compile the pi program for a three-source ISA.) Real processors don't do this yet, but research is being conducted in the area.

(*c*) Now consider the un-optimized version. Would *it* run faster with a larger reorder buffer? Should a computer designer pay attention to the performance of un-optimized code? Explain each answer.



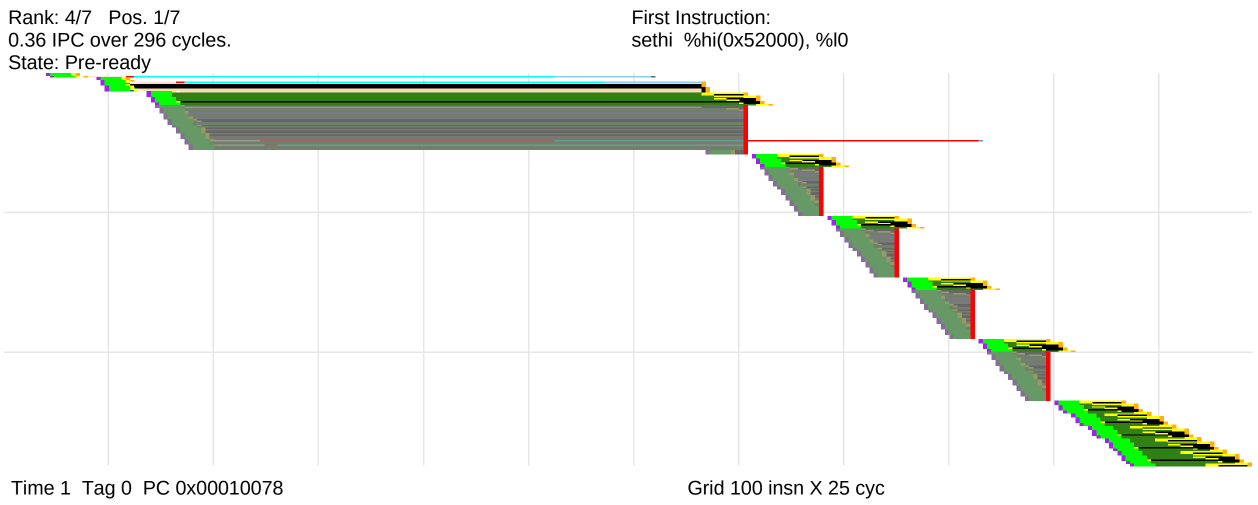
As can be seen from the PED, most instructions in the loop do not wait for dependencies, that is, they execute as soon as they are decoded, renamed, and scheduled. Some of these instructions had to wait for space in the reorder buffer, and so if the reorder buffer were larger they would be executed sooner. If some of these previously-waiting-but-now-executing-sooner-than-before instructions are on the critical path then execution speed would improve. (In the optimized case the `nop` would be executed earlier with a larger ROB but, of course, it's not on the critical path.) One way to find the critical path is to draw and analyze a dataflow graph, but an easier way is by inspection of the PED: instructions that complete just before they commit may be on the critical path. In this case the `faddd` at `0x10844`; preceded by a `fdivd`, a `ldd` at `0x838` which is dependent on the `std` at `0x898`. That store has its data ready two cycles before the load is ready to load it; the ROB was full between the store and the load, delaying the execution of the load and so increasing the ROB size will shorten this critical path.

A complete analysis shows again that there are two critical paths, this time eight cycles. (Two adds on one, an add and a sub on the other; both paths have two loads and a store. A store/load bypass takes one cycle, for a total of eight cycles.)

Since the critical paths are eight cycles and the loop spans 37 instructions the maximum IPC is 4.625. Therefore, increasing the decode width from 4- to 8-way superscalar will improve performance if the ROB has also been enlarged. (Because of the difficulty of the analysis the problem did not ask if there would be performance gain in an 8-way system.)

Computer engineers should not pay attention to unoptimized code. Most programs that are run are optimized code. (Homework assignments in beginning computer classes are an exception.) Improving the performance of un-optimized code would only help programmers debugging programs (since when debugging one usually doesn't optimize) and yes, students working on homework. If engineers' time and silicon area are limited, its better to use them to speed the kind of code most people run.

(*d*) The simulated processors use a gshare branch predictor. Use `pi_opt.ds` to answer this question. How many bits is the global history register? Entries in the PHT are initialized to 1 and the GHR is initialized to zero. The PHT is updated when the branches resolve (in the cycle after they execute). Explain your answer.



The global history register is four bits. The easiest way to find the answer is to examine execution at the start of the program, when the state of the GHR and PHT are known. Luckily, the program starts with five consecutive branch mispredictions, all for the same static instruction, `0x10820 fbl` (floating-point branch less than). Each time this branch is mispredicted not taken but is in fact taken. The PHT entries are initialized to 1 and so a particular PHT entry can only contribute to one misprediction. Therefore five PHT entries are being used. Since only one static branch is executed through these five mispredictions, the GHR contents must be four bits. Here is the GHR contents used to predict the branch the first six times: 0_2, 1_2, 11_2, 111_2, 1111_2, $1111_2 \cdots$. The fifth and sixth contents are identical, and so the branch is correctly predicted the sixth time.

(*e*) Would execution be any different if the PHT were updated when the instructions commit? Explain.

Yes. Execution is different because the PHT table is being updated after the prediction is made. For example, consider the branch that commits in cycle 253, with tag 197. The PHT would be updated at the end of cycle 253, however by that time the branch has already been predicted for the next iteration (tag 241). Had the PHT been updated at commit time the branch with tag 241 would be predicted using the old PHT value and would have been predicted not taken. Note that branches are predicted before decode and the one-cycle delay in fetching the branch target (at cycle 254) is due to another misprediction by the instruction fetch mechanism, something not covered in the class.

```
#include <stdio.h>

int
main(int argv, char **argc)
{
  double i;
  double sum = 0;                                                  // Line 7

  for(i=1; i<5000;)                                                // Line 9
    {
      sum = sum + 4.0 / i;   i += 2;                               // Line 11
      sum = sum - 4.0 / i;   i += 2;                               // Line 12
    }

  printf("After %d iterations sum = %.8f\n", (int)(i-1)/2, sum); // Line 15

  return 0;
}
```