CS231, DLDCA Lab, Lab 08

Goals

1. Performance benefits of data forwarding
2. Understanding the importance of pipeline scheduling
3. A simple but practical introduction to loop unrolling

Instructions

1. These exercises are to be done individually.
2. While you are encouraged to discuss with your colleagues, do not cross the fine line between discussion *to understand* versus discussion as a *short-cut* to complete your lab without really understanding.
3. Create a directory called <rollno>-<labno>. Store all relevant files to this lab in that directory.
    a. In the exercises, you will be asked various questions. Note down the answers to these in a file called "answers.txt".
    b. In some parts of the exercises, you will have to show a demo to a TA; these are marked as such. The evaluation for each lab will be in the subsequent lab, or during a time-slot agreed upon with the TAs. For this evaluation, you need to upload your code as well.
4. Before leaving the lab, ensure the following:
    a. You have marked attendance on SAFE
    b. You have uploaded your submission on BodhiTree, and downloaded and checked if the submissions is right
5. Things to ensure during TA evaluation of a particular lab submission:
    a. The TA has looked at your text file with the answers to various questions
    b. The TA has given you marks out of 10, and has entered it in the marks sheet
6. You have to use the MIPS conventions, unless mentioned otherwise.
7. **House points**: Some questions carry house points. For these, you are allowed to work along with others, but only if *all* of those working together have completed the lab. (You can work on it individually, even before finishing the lab completely, but only if you care about house points more than your own marks!). All questions related to house points have to be shown to the instructor (Bhaskar).

Benefits of data forwarding in the MIPS64 pipeline

- Convert the following C code into MIPS64 code in file **arraymult.s**

  double a[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

```
double b[] = {2.0, 3.0, 4.0, 5.0, 6.0, 7.0};

double c[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

double d[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

int n = 6;

double alpha = 10.0;

for (i = 0; i < n; i++) {

c[i] = a[i] * b[i];

d[i] += c[i] * alpha;

}
```

- **Question [1 mark]:** Run the MIPS64 code on WinMIPS64, with data forwarding enabled. Then subsequently run the same code with data forwarding disabled. What is the number of clock cycles taken for execution in each case. What is the speedup due to data forwarding?
- **Question [1 mark]:** If you blindly translate the C code to MIPS64 assembly code, there must be two branch instructions per loop iteration. You can optimize this to have only one branch operation per loop iteration. Do this optimization in file **arraymult-opt.s** . When forwarding is enabled, what is the speedup due to this optimization?
- From now onwards, for further modifications, we will always use data forwarding, and use the single-branch-per-iteration version of the code.

Code scheduling for performance improvement

- For this exercise, you need to be well aware of the number of cycles taken by each of the execution units in the MIPS64 pipeline implementation. You can learn this from within WinMIPS64 itself.
- Schedule the code within the loop above, such that the number of stall cycles is minimized. Write this code in file **arraymult-opt-sched.s**
- **Demo to TA [2 marks]:** Show to your TA, the unscheduled as well as the scheduled code. And show in WinMIPS64, the performance improvement due to such scheduling.
- **Demo to TA [1 mark]:** Clearly show through comments in your code, as to which are the stalls which you are not yet able to mask through scheduling.

Loop unrolling to reduce loop overhead

- Loop unrolling is a performance enhancement technique which trades off code size for code speed. That is, it achieves better execution speed at the cost of potentially larger code size. You can read briefly about loop unrolling in your textbook or online.
- You are now going to unroll the *unscheduled* code. Unroll it twice, in file **arraymult-opt-unroll2.s** . That is, two iterations of the earlier loop is now effectively achieved by one iteration of this loop. In your new loop, the loop variable will be incremented by 2 for each iteration. For simplicity, you can assume that n is a multiple of 2.

- **Demo to TA [2 marks]:** Show to your TA, the code with and without loop unrolling. And show in WinMIPS64, the performance improvement due to the unrolling.

Code scheduling after unrolling

- A non-trivial benefit of loop unrolling is the large number of opportunities it gives for better code scheduling, to avoid stalls. Do the necessary code scheduling in file **arraymult-opt-sched-unroll2.s** . To do such scheduling, you may have to do some register renaming (find out what it is) to remove anti-dependences (find out what it is).

- **Demo to TA [2 marks]:** Show to your TA, the code with and without scheduling, after unrolling. And show in WinMIPS64, the performance improvement due to the scheduling and unrolling.

- **Demo to TA [1 mark]:** Clearly show through comments in your code, as to which are the stalls which you are not yet able to mask through scheduling.

House points

- **5 house points:** Unroll the loop thrice and show the performance benefits of unrolling+scheduling in this case. How does this performance improvement compare with that due to unrolling twice? File: **arraymult-opt-sched-unroll3.s**

- **5 house points:** Change your code such that it can handle cases where *n* is not a multiple of 3. What is the performance improvement now?