

Principles in Computer Architecture

CSE 240A Homework Two

February 1, 2015

*Only Problem Set Two will be graded. **Turn in only Problem Set Two** on February 25, 2015 (**Wednesday**) by **1:00 pm**.*

*Submit homework via csemoodle <csemoodle.ucsd.edu> as a **PDF**. Other formats may or may not be viewed and evaluated accurately.*

Problem Set One

1. Hennessy & Patterson (5th Ed) 3.1
2. Hennessy & Patterson (5th Ed) 3.3
3. Hennessy & Patterson (5th Ed) 3.5
4. Hennessy & Patterson (5th Ed) 3.16
5. Hennessy & Patterson (5th Ed) 3.17
6. Hennessy & Patterson (5th Ed) 3.18
7. Hennessy & Patterson (5th Ed) B.1
8. Hennessy & Patterson (5th Ed) B.3
9. Hennessy & Patterson (5th Ed) B.9
10. Hennessy & Patterson (5th Ed) B.10

Problem Set Two

1. Rolling Registers

Rolling registers are the Next Big Thing in town! As the owner of a processor-making startup, you want to be hip and see if this new ISA feature fits your system.

The rolling registers work by introducing a *shift* instruction, which *shifts the values held by all registers*. For example, if there are 32 registers, then after a *shift* instruction, R2 holds the value of R1 from just before the shift, R3 holds the value of R2, etc. R1 holds the value of R32.

(**Part A**) Before you proceed much further, you want to evaluate the usefulness of a rolling register ISA. Rewrite the following block of code using the *shift* instruction in order to make it work better. Also tell us what function this code (and hopefully your rewrite :)) are implementing; a simple pseudo-code or behavioral description would suffice.

```
ld    R2,  #1
ld    R3,  #2
ld    R1,  #9
loop:
add   R4,  R2, R3
sub   R1,  R1, #1
add   R2,  R3, #0
add   R3,  R4, #0
bnz   R1,  loop // Repeat until R1 = 0
```

(**Part B**) Now that your boss is happy that this rolling register business seems to be delivering benefits, (s)he is getting rather aggressive trying to see if more ambitious code can be efficiently encoded with this construct. Could you help him/her by deciding **whether each of the following 4 code segments, in a loop, can be efficiently cast to use rolling registers or not**? If you decide they can, utilize the *shift* instruction and write assembly corresponding to the pseudo-code given:

1. $x[i] = x[i-2] + x[i-3]$
2. $x[i] = x[i-2] + x[i-4]$
3. $x[2*i] = x[2*i-1] + x[2*i+3]$
4. $x[2*i] = x[3*i+1] + x[4*i-1]$

(Part C) While you and your boss are involved in this idyllic experience of working out how to cast various loops to use this new construct of rolling registers, your group’s competitors in the company have sensed that something creative is afoot in your group. Knowing that creative ideas are a high risk, high reward game, they are busy going after all the risk factors in your proposal to try to nip the idea in the bud. The rival group is trying to spread rumors in the company, claiming that while some cute little loop may end up benefiting from this, the whole thing is unmanageable since once you exit the loop the register designations will be pointing to different actual registers (unless your loop iterates some multiple of the cardinality of your register file). You immediately think that perhaps a *reset* instruction will save the day, re-pointing all register references to the original value, but pretty soon you realize that this will not work, since the values will be resident at different actual register positions. You think of patching the problem up by using a shuffle network and saving the register values to the right place will work, but are concerned about the cost. You go talk to the compiler group and, being friendly and helpful, they reassure you that they can generate code that refers to the new name of the register in the block immediately following. While you appreciate their help, you are concerned about multiple dependencies in basic blocks that follow up and whether a single register name will suffice. Your boss seems confident, however, mentioning things like “we don’t have to change anything, the original names still hold in the basic blocks”. You feel better that (s)he trusts your work, but it still makes you uneasy knowing that bosses have a tendency to work with abstractions but perhaps miss a detail or two, many times crucial to the project.

Which of these ideas (or combination of ideas) do you think is the best and most efficient solution to fight the rumors that your rival group is busy spreading? Or should you admit defeat, realizing your idea is unworkable beyond the basic loop, and start a political campaign of your own to pin the technical infeasibilities as the fault of the rival group?

(**Part D**) While you are trying to figure out the pickle you find yourself in regarding register names in a sequence of loops, your spies around the company inform you of an even more inflammatory rumor that essentially boils down to “this idea won’t even work with branches!” The criticisms seem to be centered on two main parts, namely, how are the two paths of a branch, the *Taken* and the *Not Taken*, going to refer to register names and how things will work out after the branch paths re-merge if the two paths had differing number of shifts. Before you get to the political question of whether to declare this idea a failure or not, your first task is to figure out, technically, what can be done. Can you let us know so that we can give you some feedback before you let your boss know about the viability of your project?

(**Part E**) Of course, while you are trying to figure out the conceptual viability, you also need to ensure that the implementation of the idea is viable (and you thought life was going to be a bunch of roses after finishing CSE240A). You’ve consulted with some of your engineers and come to the conclusion that to implement the *shift* instruction, you need an **offset register**, indicating the **offset in the register file the current R1 is at**. However, you need to catch the rest of the company up on the strategy.

1. In which stage in the 5-stage pipeline is the *shift* instruction implemented? What happens to the offset register when the shift instruction is encountered?
2. As other instructions step through the pipeline, do they have to carry the value of the offset register associated with it?
3. Is forwarding possible with this new ISA feature, or do dependent instructions need to stall? Is interlock checking possible, or does the compiler have to add explicit NOOPs to resolve them?

2. Predictor Predilections

An important issue in current highly accurate branch predictors is that most predictors suffer from non-trivial amounts of latency, which decreases the improvement of performance enabled by high accuracy. This question asks you to explore the accuracy-latency trade-off for advanced branch prediction schemes.

(**Part A**) You have a local predictor used with the standard 5-stage pipeline. However, because of the long latency, the prediction is only available *at the end of the ID stage*, i.e. the predictor takes two cycles to return with the prediction. Therefore, the fetch continues **from the fall through path** until the ID stage ends. Then the processor will choose the path predicted by the branch predictor.

During the EXE stage, the branch condition is definitively resolved.

Assume that all the branch instructions encountered are guaranteed to be found in the Branch Target Buffer (BTB), which is accessed in the IF stage.

You benchmark your predictor, and find the prediction accuracy is 85%, and 60% of all the branches in the benchmark are taken. Please compute the average number of stall cycles per branch.

(**Part B**) In your design team, an engineer claims that the **local** predictor you are using is inferior to a new *tournament predictor*. The local predictor you currently have uses a **16-entry** table of **2-bit saturating counter** predictors, indexed by instruction PC least significant bits.

The tournament predictor being proposed has a **16-entry** table of **1-bit** predictions, indexed by instruction PC least significant bits, to choose between:

- The local predictor you currently have, or
- A **global correlating predictor** that uses a **2-bit history** to index into a table of **2-bit saturating counter** predictors. The correlating predictor is driven purely by branch history; no instruction PC bits are used in making a prediction.

Both predictions are computed for every branch. Whenever the **predictor chosen by the tournament predictor is incorrect and the other predictor is correct**, the tournament predictor flips the bit for that entry so that from **next time on, the other predictor is chosen**.

All the saturating counter predictors start in the **Strongly Taken** state (11), the tournament predictor entries are initialized to pick the local predictor, and the tournament predictor and local predictor entries are indexed by the branch address' least significant bits.

Part B 1. One part of the benchmark is given below. **Ignore branches which may be used to construct the for loop.** Do **not** worry about them neither in terms of predictor accuracy accounting nor in terms of branch history construction. The ‘then’ parts of each *if* statement is compiled to be the **Taken** side of the branch.

- For each branch, how often does the tournament predictor choose the local predictor?
- For each branch, how often does the tournament predictor choose the global predictor?
- What is the overall accuracy of the tournament predictor?
- What is the overall accuracy of just the local predictor?
- What is the overall accuracy of just the global predictor?

```
for (i = 0 to 99) {  
  
    if ((i % 4) != 0) { // Branch 1.  Address = 0x1000  
        ...  
    }  
    if ((i % 4) == 1) { // Branch 2.  Address = 0x1024  
        ...  
    }  
}
```

Part B 2. Does the predictor you selected in Part A always perform better irrespective of the program on which it is running? If yes, explain why. If not, provide a sample of code where the other predictor outperforms it and explain how.

3. Tomasulo Optimization

(Part A) For the speculative Tomasulo with reorder buffer and register renaming, one of the concerns is the number of storage elements it requires. In addition to storage for physical registers (and possibly architectural registers as well), even reservation stations need to capture the real values and hold them until all the operands have become ready. Furthermore, the ROB keeps holding the final computed value up until the entry has been committed and the instruction is graduated. One company is exploring ways to cut down on this cost by having the reservation stations no longer internally capture the values as they become ready and instead let them go to the register space (in this case the physical renamed registers) and retrieve them only once both of the values have arrived through the CDB. Of course, the physical registers cannot retain the values forever, so some kind of physical register deallocation policy (and consequently, reuse of the deallocated registers) is necessary. This vantage point has given rise to a heated debate within the company wherein a number of prominent engineers are duking it out. Some are concerned that this physical register deallocation (and reuse) will now give rise to all kinds of name hazards through the physical registers, while others are claiming that name hazards are only applicable in the case of architectural registers and are either of no concern or impossible to occur in the case of physical registers. Others are claiming that WAR hazards need to be considered, while yet others claim that the only concern that may arise is WAW hazards. Which one do you agree with and why?

(Part B) In the meantime, with all this heated debate going on, the CEO of the company, in the infinite wisdom of any CEO worth their salt, has decided to leapfrog and examine the possible implications ahead of time, in case it turns out indeed that there is reason for concern for name dependencies. (S)he has tasked a number of leading engineers in the company to see how name hazards can be handled in case they turn out to indeed be a nuisance. The engineering group has opened up their books from graduate school days and noticed that an oft-mentioned deallocation policy is to deallocate physical registers when the mapped architectural register has been rewritten. Now they are struggling to figure out what kinds of hazards may end up being encountered as a result of this deallocation policy. Some are claiming that there can be no name hazards as a result of this policy, while others claim that the only hazards that will exist will be a subset of the hazards already embedded in the architectural names. Others are claiming that there can be hazards which did not exist in the architectural registers. Another subgroup is fighting it out trying to decide which hazards one needs to worry about, the architectural register, the physical register, both or neither. Please provide your opinion as to which of these positions you agree with and provide a reasoning, perhaps complemented by code fragment and a possible reasoning to illustrate it.

(Part C) While this debate is raging on, the CEO is really interested in getting ahead of the game and has tasked a group to prepare a technique to solve any WAR hazards. Should they occur, what modifications, should one do to the pipelining in order to implement this WAR hazard obviation? Which fields should be added possibly to reorder buffer entries, reservation stations or maybe even to physical registers? Based on some of these fields, what conditions should be checked in order to ensure an orderly progression of the pipeline? (Of course, you realize quickly that instead of generating new techniques and approaches from scratch in limited time, it may be appropriate to see how techniques that you learned about during your 240A course may actually be adopted for this problem your CEO has tasked you with.)

(Part D) Continuing on in the inexorable quest for getting ahead of the game, the CEO has tasked another group to go take a look at how to handle output dependencies under this deallocation policy. Some engineers are claiming that WAW will never happen under the outlined scenario, while others claim that it can happen even more frequently and that therefore resorting to the good old scoreboard technique of simply stopping the pipeline is inadvisable. Others are claiming that stopping the pipeline is not only inadvisable but also that it will lead to deadlock with the pipeline freezing indefinitely possibly. Yet others are claiming that the only hope is to adopt the scoreboard technique verbatim and that allowing WAW hazards to proceed under this deallocation policy would wreak havoc. Please describe which of these positions you agree with and illustrate your answers with perhaps a short code fragment and a possible evolution of its fortunes through the pipeline.

(Part E) While this debate is going on, a group of Young Turks in the company is interested in staging a coup d'état and is proposing a completely different scheme, wherein the register deallocation is only to be effected once all the preceding reading registers have actually captured their values in the reservation stations that are starting to chug along. They claim that this will resolve all concerns regarding name hazards (albeit at the expense of increasing the pressure on the registers and perhaps necessitating a few extra physical registers to keep it chugging). As word of the possibly impending coup has spread in the company, the establishment is fighting back, spreading rumors regarding the workability of this technique. Depending on who is listening, the establishment claims that this technique is simply unworkable or that while workable it would necessitate so much extra information (they are mumbling something about counters to keep track of outstanding reads), or that it simply will not solve any of the name hazards anyway and so on. The Young Turks, hearing of this counterattack, and in a paroxysm of messianic revolutionary fervor are surreptitiously posting various pithy fragments, such as "Freedom or Liberty", "The answer, my friend, is blowin' in the wind", and perhaps the most damning "KISS (Keep It Simple, Stupid)". Do you think the Young Turks have a point? Can such a scheme work without necessitating the keeping of counters per physical register? If so, how? How does one deallocate registers then? Are there any WAR hazards therein? Any WAW hazards to worry about?