

CSE 240A Homework Two

April 29, 2011

*Only **Problem Set Two** will be graded. Turn in only **Problem Set Two** before May 18, 2011, 4:00pm.*

1 Problem Set One

- Hennessy & Patterson (4th Ed) 2.1
- Hennessy & Patterson (4th Ed) 2.3
- Hennessy & Patterson (4th Ed) 2.5
- Hennessy & Patterson (4th Ed) 2.7
- Hennessy & Patterson (4th Ed) 2.10
- Hennessy & Patterson (4th Ed) 2.12
- Hennessy & Patterson (4th Ed) 3.1(a)(b)(c)(d)(e)

2 Problem Set Two

1 (Branch Prediction) (36 points)

Consider the code segment below, which represents a loop containing **3** branch instructions in its body. The distance between the first two branches is exactly 16 (word address), and the distance between the last two branches is exactly 32. Assume this code segment is the only code being executed, and that the loop branch is handled **separately** and does **not** use the branch predictor. Furthermore, assume that except for the three locations shown, there exist no other references to variable **B** in the loop body.

```
for i = 0 to 15 by 1
  if (i mod 3 == 0) { // Branch1
    ...;             // Branch1 taken
    if (i mod 2 == 0) // Branch2
      B = 1;         // Branch taken
    else
      B = 0;         // Branch2 not taken
  }
  if (B == 1) {      // Branch3
    ...;             // Branch3 taken
  }
end for
```

(Part A) In this part, we are considering the use of a 2-bit saturating counter based local branch predictor. The predictor is accessed with the *least significant bits* of the branch address (word address). In order to achieve maximal prediction accuracy, what is the **minimal** number of bits of branch address that should be used? Please provide a justification for your answer. **Hint: your answer should consider the correlation between 3 branches.**

(Part B) In this part, we are considering the use of a correlating branch predictor based on pure global history with no local information. This correlating predictor is accessed with the *global branch history* that can capture **at least** the last 2 branches executed. Please comment on the usefulness of using such a correlating branch predictor in predicting *Branch3*. More specifically, how does the code layout of *Branch1* and *Branch2* impact the prediction accuracy of *Branch3*? In order to accurately predict *Branch3*, what is the minimal number of history bits that should be used? Please provide a justification for your answer.

(Part C) The code segment has been transformed into the following equivalent form.

```
for i = 0 to 15 by 1
  if (i mod 3 == 0) { // Branch1
    ...;             // Branch1 taken
    B = 1;
  }
  if (i mod 6 == 0) // Branch2
    ...;           // Branch2 taken
  B = 0;
}
if (B == 1) {      // Branch3
  ...;             // Branch3 taken
}
end for
```

What is the impact of the proposed transformation on the prediction accuracy of Branch3, compared to the initial code segment? For this code segment, in order to accurately predict *Branch3*, what is the minimal number of history bits that should be used? Please provide a justification for your answer.

(Part D) It is clear that branch prediction accuracy is crucial to the exploration of instruction level parallelism (ILP). However, highly accurate branch predictors may suffer from non-trivial amounts of access latency, which may decrease the improvement of performance enabled by high accuracy. Consider a 6 stage pipeline, **IF1 IF2 ID EXE MEM WB**, in which a branch predictor is accessed at the IF1 stage and branch conditions are definitively resolved at the EXE stage. Your job is to decide whether to use an aggressive predictor or just a simple 2-bit counter predictor. Assume these two predictors display the following characteristics:

The **aggressive branch predictor** displays a 2-cycle access latency (the fetch continues from the *fall through path* until the branch enters the **ID** stage). The prediction accuracy is 95%.

The **simple branch predictor** displays a 1-cycle access latency (the fetch starts from the predicted path when the branch enters the **IF2** stage). The prediction accuracy is 90%.

Assume that **all** the branch instructions encountered are guaranteed to be found in the Branch Target Buffer (BTB), which is also accessed in the **IF1** stage, and 60% of all the branches in the benchmark are **taken**. For each predictor, please compute the average number of stall cycles per branch and comment on its usefulness.

2 (Register Renaming) (40 points)

Instead of using a reorder buffer (ROB), one alternative approach for hardware speculation is the explicit use of an extended set of *physical registers* combined with *register renaming*. These physical registers are allocated in the issue stage to eliminate WAW and WAR hazards. Fundamentally, the architectural destination register of each instruction is mapped to a **free** physical register in the extended register set. A hardware *renaming table* is used to keep track of the mapping between two register sets.

(Part A) The following piece of code is considered for register renaming. Assume your hardware has a pool of 64 temporary registers (called *T registers* *T0* through *T63*). Additionally, assume at the initiation of the given code, *T0* through *T6* have been occupied. You are asked to perform register renaming for the code segment below, that is, to replace each destination register with the next available *T register*. (As a hint, the renaming of the first instruction has already been done for you.) Furthermore, please specify the status of the *renaming table* after renaming the **SUBD** instruction by filling in the *T registers* you have used.

```
LD    F2, 0(Rx)
LD    F4, 0(Ry)
MULTD F5, F0, F2
ADDD  F6, F0, F2
DIVD  F8, F5, F4
ADDD  F9, F0, F4
SUBD  F2, F5, F6
```

Renamed code segment:

```
LD    T7, 0(Rx)
```

Status of renaming table after renaming the SUBD instruction:

Architectural registers	F0	F2	F4	F5	F6	F8	F9
Physical registers	T4						

Unlike the rather straightforward allocation strategy, the **deallocation** of physical registers is more complicated: a physical register can only be freed up if it no longer corresponds to an architectural register **and** no further uses of the physical register are outstanding. On the other hand, the deallocation of physical registers should also be performed as early as possible, since instruction fetch will be stalled if no free physical registers are available. The remaining part of the question explores deallocation strategies. *You do not have to worry about exception behavior throughout.*

(Part B) The simplest approach that a set of engineers could come up with is “checking the source registers of all the in-flight instructions”. The manager is concerned about whether this checking works or whether it will end up in a product recall. Could you help the manager figure out the correctness of this approach? Give a brief explanation for your decision.

(Part C) Leaving the correctness of the “fully checking” mechanism aside, its high cost motivates the exploration of cheaper techniques. Consequently, another group has suggested that for an **in-order** commit processor, a physical register can be deallocated once a subsequent instruction which writes the same architectural register has been committed. Please comment on the correctness of this technique. If you think it is not correct, please provide a reasoning as to why. Otherwise, please provide a discussion of any possible shortcomings.

(Part D) The company is discussing how to handle physical register deallocation for **out-of-order** commit processors. One suggestion is to just use the deallocation strategy in (Part C), that is, to free up a physical register when a subsequent instruction writing the same architectural register has been committed. Please comment on the correctness of this technique. If you think it is not correct, please provide a reasoning as to why. Otherwise, please provide a discussion of any possible shortcomings.

(Part E) Another engineer is arguing that for out-of-order commit processors, the compiler should provide hints to the hardware in physical register deallocation. He has furthermore suggested two possible hints:

- Marking the last use of each destination architectural register.
- Providing a count of the subsequent use for each destination architectural register.

Would these hints be helpful in developing a cheaper deallocation strategy that works correctly under **any** conditions? Would these hints be helpful for some DSP codes with no branches? For each hint, if you think the hint is helpful, please give your corresponding deallocation strategy. Otherwise, please give your reasoning as to why it is not helpful.

3 (Software pipelining) (24 points)

Consider the following three loops. Each one of them reads an array value, adds a constant value to it, and subsequently stores it into another array location. A natural technique to parallelize such loops and execute them on a VLIW architecture is to apply *software pipelining*. Please assume that no memory disambiguation hardware is present and that load and store instructions can be executed in parallel but that their actual order in accessing the memory is undefined. For each of the loops below, please identify if software pipelining could be applied. (Assume that all these loops are coded as a sequence of the three load, add, and store instructions, followed by the loop overhead code.) Please explain your answers.

```
for (i=0;i<100;i++)  
    A[5*i+1]=A[2*i+3]+10;
```

```
for (i=0;i<100;i++)  
    A[2*i+1]=A[3*i+4]+20;
```

```
for (i=0;i<100;i++)  
    A[7*i+9]=A[4*i+3]+30;
```