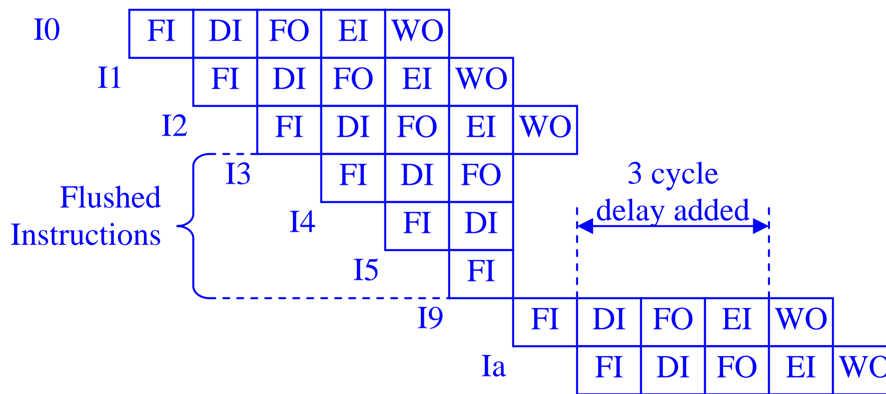
CSCI 4717 – Pipeline Homework Answers Due: Tuesday, November 21, 2005

1.) Assume that for a set of instructions, there is a 12% chance that any particular instruction is a conditional branch instruction. Of that 12%, one fourth of them result in a branch to a nonconsecutive address, i.e., the pipeline will have to be flushed. Ignoring any branch prediction algorithm, and using τ to represent the time it takes to execute a single stage of the pipeline, predict how long it will take to execute 1200 instructions. Assume the pipeline has 5-stages.

A conditional jump adds cycles to the execution of code by forcing the pipeline to be flushed. Now, assuming the pipe is full during each one of these flushes, a certain number of cycles will be added to the execution of code. Below is a sample execution of instructions in a 5-stage pipeline where instruction I2 is a "taken" branch to instruction I9 that required a flush of the pipeline.



If 12% of the instructions are branch instructions, and if one fourth of them result in a branch, then 12% × ¼ = 3% of the 1200 instructions cause the pipeline to be flushed. Looking at the figure above, we see that a pipeline flush pushes the WO stage out resulting in 3 cycles (k–2 cycles where k=5) being inserted into the overall duration of the stream. (I accepted 3 or 4 cycles added per flush in your answers.)

Enough of the general discussion, let's move on to the calculation. Without any branches, the number of cycles needed to execute 1200 instructions in a 5-stage pipeline (k=5) would be calculated as shown below. (k = number of stages, n = number of instructions)

$$\begin{aligned} \text{number of cycles wo/flush} &= k - 1 + n \\ &= 5 - 1 + 1200 \\ &= 1204 \text{ cycles} \end{aligned}$$

The number of cycles added due to a flush equals the number of flushes multiplied by k – 2. (Once again, I also accepted k – 1.)

$$\begin{aligned} \text{number of cycles added for flush} &= (0.03 \times 1200) \times (k - 2) \\ &= 36 \times 3 \\ &= 108 \text{ cycles} \end{aligned}$$

Finally, the time required to execute this sequence of instructions is τ times the number of cycles.

$T = \tau \times$ [number of cycles/stages added for flush + number of cycles/stages wo/flush]

$T = \tau \times [1204 + 108] = \tau \times 1312$

2.) Consider the following section of code.

```
for (i=0; i<10; i++)
{
    for (j=0; j<5; j++)
    {
        <code containing no conditional jumps>
    }
}
```

a.) Once compiled, how many conditional jumps would be contained in the above section of code? (static occurrence)

There should be a conditional jump at the end of each loop to check if the final condition has been exceeded. Therefore, there are ***2*** static occurrences of the conditional jumps.

b.) After fully executing the above section of code, how many conditional jumps would the CPU have encountered? (dynamic occurrence)

The outer loop is executed 10 times. Therefore, the conditional jump for the outer loop is encountered 10 times. The inner loop is executed 5 times for every execution of the outer loop, i.e., it is executed $5 \times 10 = 50$ times. Therefore, there are ***10 + 50 = 60*** dynamic occurrences of the conditional jumps.

c.) Using the static branch prediction algorithm "branch always," how many of the conditional jumps calculated in the previous problem would have been predicted incorrectly?

For every completed operation of a loop, there will be one encounter where the conditional jump is ***not taken***. This is when the final condition has been exceeded. Since we "fall out" of the outer loop once and we "fall out" of the inner loop 10, then there are ***10 + 1 = 11*** times when the branch would not have been taken and therefore predicted incorrectly.

d.) Using the branch prediction algorithm described in figures 12.16 and 12.17, how many of the conditional jumps calculated in part b would have been predicted correctly assuming an initial state of "predict taken"?

If you look at the state diagram in Figure 12.17, you should see that there must be two incorrect predictions before the system will change its prediction. The only time that the outer loop has a failed prediction is at the very end of the operation of the code segment shown, and therefore, there is no chance for a second incorrect prediction to change the system from "predict taken" to "predict not taken." The inner loop has an incorrect prediction at the end of its five loops. The next time the conditional jump is encountered is when the loop has been started again, and at that point, the branch is taken and the prediction is correct. Therefore, the system will once again always predict that the branch is to be taken and the answer is the same as for problem , i.e., ***11*** incorrect predictions and ***49*** correct predictions.

3.) Modify the following piece of code in order to support delayed branching and delayed loading. Assume a load from memory will force a subsequent instruction to stall in the pipeline if it uses the same register.

Delayed branching: A NOP needs to be placed at the end of every branch in order to avoid flushing the pipeline. The code below has just one branch, a conditional one immediately after the compare.

Delayed loading: A NOP needs to be placed between any two instructions where data dependency occurs, i.e., where a register is operated on (written to) and then immediately used as an input (read from) for the next instruction. This occurs in two places, right before AL is stored and when As for

```
     MOV   CL,0             ;Initialize counter CL

L1:  MOV   AL,[BX:1000]     ;Retrieve data from address 1000

                            ;It could be said that a NOP is justified
                            ;here, but it is unlikely that it would be
                            ;needed.

     ADD   AL,23            ;Add immediate value

     NOP                    ;NOP to allow AL to settle before storing

     MOV   [BX:1000],AL     ;Store result back to address 1000

     INC   CL               ;Increment CL

     NOP                    ;NOP to allow CL to settle before comparing

     CMP   CL, 100          ;Check to see if CL equals 100

     JNE   L1               ;If not, continued looping

     NOP                    ;NOP to avoid flush after jump

     MOV   AL,1             ;Storing 1 at address 1001...

     NOP                    ;NOP to allow AL to settle before storing

     MOV   [BX:1001],AL     ;...indicates we're done
```