EE 4720 **Homework 5** Solution **Due: 17 November 2000**

Problem 1: The familiar loop below executes on a dynamically scheduled machine using a reorder buffer to name destination registers. The machine has the following characteristics:

- Two-way superscalar. An unlimited number of write-backs per cycle.
- A 16-entry reorder buffer.
- A six-stage fully pipelined floating point multiply unit.
- Perfect branch target prediction. (Branch target in IF when branch is in ID.)

Show a pipeline execution diagram up to the fetch of the third iteration.

Explain why the first two iterations cannot be used to determine the CPI for a large number of iterations in this case. Estimate the CPI for a large number of iterations (a pipeline execution diagram is not necessary).

```
LOOP: ! LOOP = 0x1000
! Cycle              0  1  2  3  4  5  6  7  8  9  10 11
 ld   f0, 0(r1)      IF ID L1 L2 WB    IF ...
                              IF ID L1 L2 WB
 muld f2, f0, f2     IF ID RS    M1 M2 M3 M4 M5 M6 WB
                              IF ID RS             M1 M2 M3 M4 M5 M6 WB
                                    IF ...
 addi r1, r1, #8        IF ID EX WB
                                 IF ID EX WB
 sub  r2, r1, r3        IF ID RS EX WB
                                 IF ID RS EX WB
 bneq r2, LOOP             IF ID RS B  WB
                                    IF ID RS B  WB
 xor  r10, r11, r12        IF x     IF x

 and  r13, r14, r15
 or   r16, r17, r18
 sgt  r19, r20, r21
```

For clarity the first iteration is shown in black, the second in blue, and the third (just IF's) in orange. The first two iterations can't be used to determine CPI because they start differently, for example, in the first `f2` is available, but at the beginning of the second (cycle 3) the value for `f2` is not yet ready.

The CPI for a large number of iterations would be limited by the multiply unit. The hardware can fetch and decode at a rate of 3 cycles per iteration, but the multiply latency is 6. Because there is a loop-carried dependency on the multiplier input the multiplies have to be done one after another, and so execution is limited to 6 cycles per iteration (after the reorder buffer fills). Since there are five instructions in an iteration the CPI is limited to $\frac{6}{5}$.

Problem 2: Unroll the loop in the problem above twice. (In the last homework it was unrolled four times.) Again exploiting the associativity of multiplication, rearrange the multiplies to improve the performance, but this time without using software pipelining. Why is software pipelining not necessary here?

```
! Solution
LOOP: ! LOOP = 0x1000
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
 ld   f0, 0(r1)   IF ID L1 L2 WB          IF ID L1 L2 WB
                               IF ID L1 L2 WB          IF ...
 ld   f10, 8(r1)  IF ID RS L1 L2 WB       IF ID RS L1 L2 WB
                               IF ID RS L1 L2 WB
 muld f4, f0, f10    IF ID RS    M1 M2 M3 M4 M5 M6 WB
                                  IF ID RS    M1 M2 M3 M4 M5 M6 WB
                                              IF ID RS    M1 M2 M3 M4 M5 M6 WB
 muld f2, f4, f2     IF ID RS                       M1 M2 M3 M4 M5 M6 WB
                                  IF ID RS                          M1 M2 M3 M4 M5 M6 WB
                                              IF ID RS                                M1
 addi r1, r1, #16       IF ID EX WB IF ID EX WB IF ID EX WB
 sub  r2, r1, r3        IF ID RS EX WB          IF ID RS EX WB
                                  IF ID RS EX WB
 bneq r2, LOOP             IF ID B  WB IF ID B  WB IF ID B  WB
 xor  r10, r11, r12        IF x        IF x        IF x
 and  r13, r14, r15
 or   r16, r17, r18
 sgt  r19, r20, r21
```

For clarity the first iteration is shown in black, the second in blue, the third in orange, and the fourth (just an IF) in purple. (A pipeline diagram was not required for the solution, but is given here to help describe the solution.)

An important feature of the solution is the way the multiplies are done. The code above is limited to execute at a rate of six cycles per iteration because of the loop-carried dependency in the second multiply. (But this does twice as much work as the original code.) In the poor solution below the code is half as fast, limited to twelve cycles per iteration because the loop-carried dependency is a source in the first multiply and a destination in the second:

```
! WARNING: POOR Solution below!
LOOP: ! LOOP = 0x1000
 ld   f0, 0(r1)
 ld   f10, 8(r1)
 muld f2, f0, f2
 muld f2, f10, f2
 addi r1, r1, #16
 sub  r2, r1, r3
 bneq r2, LOOP
 xor  r10, r11, r12
 and  r13, r14, r15
 or   r16, r17, r18
 sgt  r19, r20, r21
! WARNING: POOR Solution above!
```

Refer to the good solution for the following discussion.

Software pipelining is not needed because dynamic scheduling allows instructions after the second multiply to start execution even before the second multiply starts. On a statically scheduled machine instructions after the second multiply would have to wait. Software pipelining can be used to reduce the wait by moving the second multiply to the next iteration.

Problem 3: The code below executes on a system using a one-level branch predictor with a 16-entry BHT. Which entries will the branches use?

The BHT entry numbers are shown in the leftmost column below. The entry numbers are bits 2:5 in the instruction address, shown in the second column.

If the number of iterations is large, the prediction accuracy will be high. If a certain number of additional nops are inserted before `SKIP1` the prediction accuracy will drop. How many and why?

By inserting `nop` instructions the BHT entry used by the second and third branches will change. Prediction accuracy will fall if the first and second branch use the same entry since their outcomes are always different from each other. Each inserted `nop` increases the BHT entry number by one, 13 `nop`'s would put the second branch in entry 1, the same as the first.

```
! Note: r2 is not modified inside the loop.
BHT En   Addr    LOOP:  ! LOOP = 0x1000
         0x1000:  subi r1, r1, #1
 1       0x1004:  bneq r2, SKIP1
         0x1008:  add r10, r10, r11
         0x100c:  nop
                 SKIP1:
 4       0x1010:  beqz r2, SKIP2
         0x1014:  add r12, r12, r13
                 SKIP2
 6       0x1018:  bneq r1, LOOP
```

Problem 4: Determine the prediction accuracy of a one-level branch predictor on each branch in the code below. The predictor uses a 1024-entry BHT. There is a .5 probability that a loaded value will be zero.

Because random numbers are loaded, the first branch (following `LOOP`) and the branch following `SKIP2` can't be predicted, so the accuracy will be about 50%.

The second branch (following `SKIP1`) follows the pattern `N T N T ...`. Depending on how the `BHT` entry is initialized, the prediction accuracy will be 50% or 0%.

The third branch (following `SKIP3`) follows the pattern `N T T T N T T T ...`. The prediction accuracy will be 75% (the not taken is predicted taken after warm up).

The last branch (following `SKIP4`) is taken for all but the last iteration, the prediction accuracy will be 100% for branches predicted after the first two iterations of the loop.

```
LOOP:
 addi r2, r2, #4
 lw r1, 0(r2)
 bneq r1, SKIP1
 add  r10, r10, r11
SKIP1:
 andi r3, r2, #4
 bneq r3, SKIP2
 add  r11, r11, r12
SKIP2:
 beqz r1, SKIP3
 add  r12, r12, r11
SKIP3:
 andi r4, r2, #12
 bneq r4, SKIP4
 add  r13, r13, r11
SKIP4:
 sub r5, r2, r6
 bneq r5, LOOP
```

Problem 5: How many BHT entries will the branches in the code above use in the middle of its execution (explained below) in a two-level gselect predictor that uses 10 bits of global branch history and 6 instruction address bits? The loop iterates many times, the middle of its execution starts after many iterations.

The global history has the following repeating pattern:
`rNrNT rTrTT rNrTT rTrTT rNrNT rTrTT rNrTT rTrTT ...`, where `r` is random and can be either `T` or `N`. Each group corresponds to an iteration. The global history register contains ten outcomes. The global history when predicting the first branch in the loop might see `rNrNT rTrTT`, the global history for the second branch might see `NrNT rTrTT r`, and so on.

Ignoring the `r`'s, each branch can see four possible global history patterns (since there are four sets of branch outcomes in an iteration such as `rNrNT` and they occur in the same order each time). Taking the global history into account, there are 16 variation on each pattern (since each pattern contains 4 `r`'s). Therefore each branch can see 64 different patterns. There will be a different BHT entry for each branch and each pattern (since there are no collisions) and so the total number of BHT entries is $16 \times 4 \times 5 = 320$.

How many bits of global branch history are needed so that the branch following `SKIP3` is predicted very accurately?

The branch following `SKIP3` follows the pattern `N T T T N T T T ...`. To distinguish the not taken case from the others the branch predictor might look at the three previous outcomes of the `SKIP3` branch. If they are all taken it would predict not taken. That would require a global history length of 15. However, it's possible to use a shorter global history: look at the two previous outcomes of the `SKIP3` and the `SKIP1` branch. If the two last `SKIP3` branches are `TT` and the two last `SKIP1` branches are `TN`, predict not taken. (Don't forget that the global history contains all branches in this loop, but the other branches here are just noise.) So the minimum global history size is just 10 outcomes.