EE 4720 Homework 3 Solution Due: 15 March 2000

Problem 1: Show a pipeline execution diagram for the following DLX code fragment on a statically scheduled implementation in which the add functional unit has a latency of 3 (four stages) and an initiation interval of 2 (not the usual 1) and the multiply unit has a latency of 5 (six stages) and has an initiation interval of 3 (not the usual 1).

```
 ! Solution
 ! Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 addf f0, f1, f2    IF ID A1 A1 A2 A2 WB
 addf f3, f0, f4       IF ID -------> A1 A1 A2 A2 WB
 addf f5, f0, f7          IF -------> ID -> A1 A1 A2 A2 WB
 gtf  f0, f8                          IF -> ID -> A1 A1 A2 A2 WB
 multf f9, f0, f10                          IF -> ID M1 M1 M1 M2 M2 M2 WB
```

Problem 2: The following DLX code fragment executes on a statically scheduled implementation in which the add functional unit has a latency of 3 (four stages) and an initiation interval of 1 (the usual 1) and the multiply unit has a latency of 5 (six stages) and has an initiation interval of 1 (the usual 1).

The implementation uses ID-stage branch target calculation. As is true for the pipelines used in class, the branch condition is not bypassed.

Instructions stall in ID to avoid structural hazards.

There are bypass paths from the WB stage to the inputs of the floating-point functional units.

(*a*) Show a pipeline execution diagram for the code.

```
LOOP:   ! Solution
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 multd f0, f0, f2  IF ID M0 M1 M2 M3 M4 M5 WB          IF ID
 ld    f4, 0(r1)      IF ID EX ME WB                      IF
 addd  f2, f2, f4        IF ID -> A0 A1 A2 A3 WB
 addi  r1, r1, #8           IF -> ID ----> EX ME WB
 sub   r2, r1, r3                 IF ----> ID EX ME WB
 bneq  r2, LOOP                         IF ID ----> EX ME
 xor   r10,r11,r12                         IF ----> x
```

(*b*) What is the CPI for a large number of iterations of the loop?

The first iteration takes 12 cycles. The state of the pipeline at the beginning of the second iteration (cycle 12) is different then the state at the beginning of the first (cycle 0) because the branch instruction from the first iteration is still present. That branch instruction finishes at the end of cycle 14 and will not change the way the second iteration executes, and so the second iteration will also take 12 cycles. Therefore the CPI for a large number of iterations is $\frac{12}{6} = 2$ cycles per instruction.

(*c*) If the multiply functional unit latency were long enough the second iteration would take longer than the first iteration. (An iteration starts when the first instruction is in IF.) What is the smallest such latency?

The multiply uses values produced in a previous iteration (that is, it has a *loop-carried dependency*). If those values aren't ready execution will stall. In the example below the execution of multiply in the second iteration is stalled for one cycle (at cycle 15) because the result from the previous iteration is not ready. In this example the multiply unit has a latency of 13 cycles, is the latency were 12 cycles there would be no stall, and so the smallest latency that will increase the duration of the second iteration is 13 cycles.

```
! Part of Solution
! Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 multd f0, f0, f2  IF ID M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10M11WB
                                                      IF ID -> M0 M1
 ld    f4, 0(r1)      IF ID EX ME WB
 addd  f2, f2, f4        IF ID -> A0 A1 A2 A3 WB
 addi  r1, r1, #8           IF -> ID EX ME WB
 sub   r2, r1, r3                 IF ID -> EX ME WB
 bneq  r2, LOOP                      IF -> ID ----> EX ME -> WB
 xor   r10,r11,r12                         IF ----> x
```

Problem 3: Schedule—but don't unroll—the code from the problem above to avoid as many stalls as possible. Show a pipeline execution diagram of the scheduled code. *Hint: you can change the offset of the load double instruction.*

```
LOOP: ! Solution
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13
 addi  r1, r1, #8 IF ID EX ME WB       IF ID -> EX ME WB       IF ID -> EX ME WB
 sub   r2, r1, r3    IF ID EX ME WB       IF -> ID EX ME WB
 ld    f4, -8(r1)       IF ID EX ME WB          IF ID EX ME WB
 multd f0, f0, f2          IF ID M0 M1 M2 M3 M4 M5 WB
                                                     IF ID M0 M1 M2 M3 M4 M5 WB
 addd  f2, f2, f4             IF ID A0 A1 A2 A3 WB    IF ID A0 A1 A2 A3 WB
 bneq  r2, LOOP                  IF ID EX ME WB          IF ID EX ME WB
 xor   r10,r11,r12                  IFx                     IFx
```

Problem 4: Unroll the loop below so that two iterations of the original loop form one unrolled loop. Schedule the code so that it executes as efficiently as possible. Assume there will be an even number of iterations and that every register not used in the original code is available and so can be used in the unrolled loop. The loop runs on the implementation described in the second problem.

```
LOOP:
 ld     f0, 0(r1)
 multd  f0, f0, f2
 addd   f0, f0, f4
 sd     8(r1), f0
 addi   r1, r1, #16
 sub    r2, r1, r3
 bneq   r2, LOOP
```

Two solutions are provided below. In the first the loop is unrolled without software pipelining. That is, the work done by one iteration of the unrolled loop is exactly the work done by two iterations of the original loop. This solution has several stall cycles, as can be seen in the pipeline execution diagram.

The second solution also uses software pipelining. A single iteration of this loop does the work of four half-iterations of the original loop. Instructions `addd f6, f5, f4` and `sd 8(r1), f6` are part of one half-iteration, `addd f16, f15, f4` and `sd 8(r1), f16` are part of another half-iteration, `ld f0, 32(r1)` and `multd f5, f0, f2` are part of a third half-iteration, and `ld f10, 32(r1)` and `multd f15, f10, f2` are part of a fourth half-iteration. This solution suffers no stalls, it only looses a cycle due to the branch penalty. In class, software pipelining was covered in the context of IA-64 register rotation, but as shown below it can also be used with conventional ISAs.

```
! Solution 1: Unrolled, but no software pipelining.
LOOP:
 ld     f0, 0(r1)       IF ID EX ME WB
 ld     f10, 16(r1)        IF ID EX ME WB
 multd  f0, f0, f2            IF ID M0 M1 M2 M3 M4 M5 WB
 multd  f10, f10, f2             IF ID M0 M1 M2 M3 M4 M5 WB
 addi   r1, r1, #32                 IF ID EX ME WB
 sub    r2, r1, r3                     IF ID EX ME WB
 addd   f0, f0, f4                        IF ID ----> A0 A1 A2 A3 WB
 addd   f10, f10, f4                         IF ----> ID A0 A1 A2 A3 WB
 sd     8(r1), f0                                     IF ID -------> EX ME WB
 sd     24(r1), f10                                      IF -------> ID EX ME WB
 bneq   r2, LOOP                                                     IF ID EX ME
```

```
! Solution 2:
! Unrolled loop with software pipelining.  No stalls.

! Prologue
 ld     f0, 0(r1)
 multd  f8, f0, f2
 ld     f0, 16(r1)
 multd  f18, f0, f2
 subi   r13, r3, #32  ! In loop position of addi and sub swapped.

! Cycle                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
LOOP:
 addd   f6, f8, f4     IF ID A0 A1 A2 A3 WB                IF ID A0 A1 A2 A3
 ld     f0, 32(r1)        IF ID EX ME WB                      IF ID EX ME WB
 addd   f16, f18, f4         IF ID A0 A1 A2 A3 WB                IF ID A0 A1
 ld     f10, 48(r1)             IF ID EX ME WB                      IF ID EX
 multd  f8, f0, f2                 IF ID M0 M1 M2 M3 M4 M5 WB          IF ID
 sub    r2, r1, r13                   IF ID EX ME WB                      IF
 sd     8(r1), f6                        IF ID EX ME WB
 sd     24(r1), f16                         IF ID EX ME WB
 multd  f18, f10, f2                           IF ID M0 M1 M2 M3 M4 M5 WB
 addi   r1, r1, #32                               IF ID EX ME WB
 bneq   r2, LOOP                                     IF ID

! Epilogue

 addd   f0, f8, f4
 sd     8(r1), f0
 addd   f0, f18, f4
 sd     24(r1), f0
```