CMSC 411
Homework #4 Solutions (partial)

Exercise 4.1 There are seven dependences in the C loop presented in the exercise:

1. True dependence from S1 to S2 on a.
2. Loop-carried true dependence from S4 to S1 on b.
3. Loop-carried true dependence from S4 to S3 on b.
4. Loop-carried true dependence from S4 to S4 on b.
5. Loop-carried output dependence from S1 to S3 on a.
6. Loop-carried antidependence from S1 to S3 on a.
7. Loop-carried antidependence from S2 to S3 on a.

For a loop to be parallel, each iteration must be independent of all others, which is not the case in the code used for this exercise.

Because dependences 1, 2, 3, and 4 are "true" dependences, they can not be removed through renaming. In addition, as dependences 2, 3, and 4 are loopcarried, they imply that iterations of the loop are not independent. These factors together imply the loop cannot be made parallel as the loop is written. By "rewriting" the loop it may be possible to find a loop that is functionally equivalent to the original loop that can be made parallel. Exercise 4.2 provides an example of such a situation on a different loop.

Exercise 4.2 There are five dependences in the C loop presented in the exercise:

1. True dependence from S1 to S2 on a.
2. Anti-dependence from S1 to S2 on b.
3. Loop-carried true dependence from S3 to S2 on a.
4. Loop-carried true dependence from S3 to S3 on a.
5. Loop-carried output dependence from S3 to S1 on a.

To parallelize the loop, we simply "break" all of the loop-carried dependences. The loop-carried output dependence can be removed by the compiler through renaming while the loop-carried true dependences can be removed if the compiler modifies the code.

The C loop in the exercise can be re-written as shown in Figure 4.1 to make it parallel. In this case, this can be accomplished by removing statement S3 from the loop. Looking at the original code, it should be apparent the S3 does not do any useful work in the be apparent the S3 does not do any useful work in the loop. Although S3 assigns a value to a[i+1], this value is over-written by S1 on the next iteration before the value written by S3 is used. The transformed version of the loop shown in Figure 4.1 is functionally equivalent to the original loop from the exercise.

```
for (i=1; i<=100; i=i+1) {
    a[i] = b[i] + c[i];  /* S1 */
    b[i] = a[i] + d[i];  /* S2 */
}
```

Figure 4.1: Parallel Version of the Loop

Exercise 4.4 In the exercise, we are asked to unroll the loop and schedule it for a pipelined version of DLX. We assume that the loop was originally executed a non-zero, even number of times, otherwise, more sophisticated transformations would be required. We have also schedule the branch-delay slot.

When unrolling a loop with no loop-carried dependences, one can follow some basic guidelines. First, copy all the statements in the original loop and put them after the original loop statements. Second, rename all the registers in the copied instructions so that they are distinct from the original statements (this can be done by adding a fixed value to each register number, assuming there are enough registers available). Third, interleave all of the statements by putting the i'th instruction in the original sequence. These steps yield a schedule without violating data dependences.

One can then remove loop overhead instructions and rearrange other instructions as necessary to cover pipeline latencies. Instructions that use or update index calculations will have to be updated based on reordering of instructions or elimination of intermediate index updates.

Doing these steps and reordering instructions to cover any remaining latencies, yields the code shown in Figure 4.2 . In this code, the comments indicate the amount of latency and the instruction from which the latency is measured (e.g., ">1 from LD F0,0(R1)" indicates the instruction must follow the specified load by more than one cycle).

```
loop:   LD F0,0(R1)      ;
        LD F6,-8(R1)     ;
        MULTD F0,F0,F2   ; >1 from LD F0
        MULTD F6,F6,F2   ; >1 from LD F6
        LD F4,0(R2)      ;
        LD F8,-8(R2)     ;
        ADDD F0,F0,F4    ; >3 from MULTD F0
        ADDD F6,F6,F8    ; >3 from MULTD F6
        SUBI R1,R1,16    ;
        SUBI R2,R2,16    ;
        SD 8(R2),F0      ; >2 from ADDD F0
        BNEZ R1,loop     ;
        SD 0(R2),F6      ; >2 from ADDD F6,
                         ;  fills branch delay
```

Figure 4.2: Code Unrolled Once and Scheduled

Exercise 4.5 For the solution below, we assume that this loop will be executed a nonzero, even number of times. We have also scheduled the branch-delay slot and follow the same basic rules to unroll the loop described in the solution to Exercise 4.4. The main difference from Exercise 4.4 is that there is a loop carried dependence in this loop. Due to the latencies of the FPU, and small number of instructions in this loop (especially after removing extra SUBI's), unrolling is not effective all by itself.

The solution shown in Figure 4.3 takes advantage of the associativity of sum reduction and computes all the even indices into one register and all the odd indices into another. By doing this, the instructions from two consecutive iterations of the original loop can be freely scheduled together. After the loop body, the registers holding the even and odd streams are added to get the final result. We assume that registers F2 and F10 are zero prior to beginning the loop. The comments in this figure indicate the amount of latency being covered and the source of the latency (e.g., "3 from MULTD F0,F0,F4" indicates that three cycles of latency are being covered from the indicated multiply).

```
loop:
        LD F0,0(R1)      ;
        LD F6,8(R1)      ;
        LD F4,0(R2)      ;
        LD F8,8(R2)      ;
        MULTD F0,F0,F4   ; 1 from LD F4,0(R2)
        MULTD F6,F6,F8   ; 1 from LD F8,8(R2)
        SUBI R1,R1,16    ;
        SUBI R2,R2,16    ;
        ADDD F2,F0,F2    ; 3 from MULTD F0,F0,F4

        BNEQZ R1,loop    ;
        ADDD F10,F6,F10  ; 3 from MULTD F6,F6,F8
                         ; fill branch delay
out_of_loop:
        ADDD F2,F10,F2   ; combine even-odd
                         ; streams
```

Figure 4.3: Unrolled and Scheduled Code