MAKE SURE THAT YOU DO NOT CREATE UNWANTED EXCEPTIONS.

Problem 3.25

We revisit the code of Problem 3.24 above with branches and jumps.

a. Identify all basic blocks in the code

b. Schedule the code the best you can by using local scheduling only (i.e., within basic blocks) on the VLIW machine of Problem 3.22. Note that the branch is delayed by one cycle. This delay has not been scheduled in the code.

c. A simple compiler designed by a student in a computer science class project reorganizes the code globally as follows, thinking that it will have better performance for the same result (in this code again branch delays have not been taken into account).

```
I1              LW R5,0(R1)              /load A
I2              LW R7,0(R3)              /load C
I3              SLT R8,R5,R7
I11             LW R11,0(R4)             /load D
I12             SLT R12,R11,R6
I14             ADD R13,R7,R11
I5              LW R6,0(R2)              /load B
I6              SLT R9,R5,R6
I8              ADD R10,R6,R7
I16             SUB R14,R5,R7
I4              BNEZ R8, then            /test A<C
I7              BNEZ R9, then            /test A<B
I9              SW R10,0(R1)             /execute if clause
I10             J exit
I13     then    BNEZ R12,then1           /test D<B
I15             BNE R5,R13,exit          /test A==C+D
I17     then1   SW R14,0(R2)             /execute then clause
I18     exit
```

The student argues that this code is correct because the result in memory will always be the same as for the original code. However, the students ignores a few things:
--LW instructions were moved up across stores, which creates memory hazards
--instructions causing exceptions were moved up across jumps and branches, which creates control hazards and possible unwanted exceptions.

At least the student decided not to move stores up across branches (why is this a good thing?).

In any case a mechanism is needed to detect and correct the problems with memory hazards and exceptions. For this purpose, we use mechanisms and instructions similar to those in the IA-64 ISA.

First we deal with exceptions. Assume first that no instructions raises exceptions except for loads and stores. Remember that stores can never be speculative). When a load is elevated across a branch (with its dependent instructions) it becomes a speculative load with opcode LW.s (e.g., LW.s R1,0(R2)). Because the load is now speculative and its execution may not be required, no exception that is visible to the program may be signalled. (For example a page fault is not visible to the program, but an address misalignment is.) When such an exception occurs on a speculative load, the value returned by the load is often undefined. Thus the destination register is poisoned and its

content is replaced by an exception descriptor. At the location where the LW was in the original code, a check instruction is inserted, with format check.s R1,repair. If the speculative load and its dependent instructions were not supposed to be executed, then the check.s instruction is not executed. On the other hand if the check.s instruction was supposed to be executed, the check.s instruction is executed and looks up the register. If the register is valid, all is good and execution proceeds. If the register is poisoned, then the execution jumps to repair code which essentially re-execute the sequence of elevated instruction, but without the speculative load. At this time the exception is taken.

Second, we deal with memory hazards, using a similar mechanism. When a load is elevated with its dependent instructions across one or multiple stores and the compiler cannot disambiguate addresses, the load value becomes speculative and the load becomes LW.a (e.g., LW.a R1,0(R2)). At the location where the load was in the original code a check instruction is inserted with format check.a 0(R2),repair. This instruction works in conjunction with the ALAT: the LW.a inserts its address in the ALAT; if a store with the same address is executed between the LW.a and the check.a, the address is removed from the ALAT and the check.a can detect this. In case the value returned by the LW.a was stale, the check.a instruction jumps to some fix-up code, which mostly repeats the load and its dependent instructions. If a load and its dependent instructions are moved up across stores and branches, the speculative load becomes LW.as R1,0(R2) and only a check.a instructions is needed because when a check.as instruction gets an exception, no address is stored in the ALAT.

Do the following:

--Add the instructions in the new code to check for mispeculations (both memory and exceptions)
--schedule the new code locally, taking advantage of the branch and jump delay slots
--schedule the code on the VLIW machine of Problem 3.22.

What is the speedup obtained by this new code on the VLIW machine, for all possible cases?
When is the new code with speculative loads better?

Problem 3.26

Vector processors need fast scalar processors to fight Amdhal's law by running the code that cannot be vectorized as fast as possible. One very common vector operation is the dot-product of two vectors, which is a scalar. The dot-product is the basic operation in matrix multiply and most signal filtering operations. The dot-product of two vectors X and Y of dimension n is:

$$X \cdot Y = \sum_{k=1}^{n} x_k y_k$$

The corresponding C code is:

```
for(k=0; k<n; k++) p += x[k]*y[k];
```

The problem with this code is that it has a loop carried dependency. However it can still be computed efficiently on a vector processor backed up by a high-performance scalar processor. There are two operations in the dot-product: the multiplication of two vectors followed by the accumulation of the components of the result.

To do this, the loop is strip-mined in slices of 64 components and the two input vector slices are