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

multiplied and the result of this multiplication is accumulated in a result vector register. After all the slices have been processed, the 64 components of the result vector register are added together to form the dot product. To speedup the processing of each slice, vector operations are chained and run in parallel.

a. Assume an architecture similar to the one in Table 3.42, with 8 vector registers of 64 components each, and a very large number of memory banks (say 1024) so that conflicts for banks never happen. The bank access time is 30 cycles, the stride is 1, the latency of the multiply pipeline is 10, and the latency of the add pipeline is 5.
Give the code for the processing of each 64 component slice using vector loads and arithmetic instructions as in Section 3.7. Compute the time taken by a dot-product where the vector sizes are 1024 each (neglect the final scalar phase to accumulate the components)

b. How many clocks does it take to compute the multiplication of two 1024x1024 matrices using the same algorithm (neglect the final scalar phases)?

c. We can unroll the vector loop twice (there are enough vector registers). Show the code for the dot-product and calculate the number of clocks needed to compute the multiplication of two 1024x1024 matrices (neglect the final scalar phases).

Problem 3.27

In this chapter, we have assumed a simple interleaved memory system for a vector processor, with four banks, a bank access latency of 4 and a stride of 1. However, in practice there will be more banks and moreover the stride may vary to access complex structures.

a. Assume 32 banks, a bank access time of 8 clocks, and one vector load or one vector store at a time. What are the permissible vector strides (from 1 to 32) that will avoid conflict, using the simple interleaved scheme in Figure 3.41. Can you generalize this result to any stride by considering the stride modulo 32?

b. Assume now 31 banks, a bank access time of 8 clocks and one vector load or one vector store at a time. What are the permissible vector strides (from 1 to 31) that will avoid conflict, using the simple interleaved scheme in Figure 3.41. Can you generalize this result to any stride by considering the stride modulo 31?

c. Discuss the advantages and inconveniences in using 31 banks vs. 32 banks.

Problem 3.28

Not all vector machines are load/store. In fact the first vector machine, the CDC Star-100 (built at the beginning of the 1970s by a now defunct company called Control Data Corporation) was a pure memory-to-memory vector machine. Input vectors were streamed from memory directly into pipelined units and the results were streamed directly back to memory. This style of architecture was very memory intensive and resulted in huge delays to move data back-and-forth to memory. Although the CDC Star-100 was followed up at the beginning of the 1980's by the CDC Cyber205 this class of machine was eventually supplanted by Cray machines which were load/store vector machines (starting with the Cray-1 in 1976). In the Cyber-205 vector from memory could be as