Problem 1

Branch prediction is guessing which path a conditional branch will take and executing as much of it as can be executed before the condition deciding the branch is computed. It also means that the code which is executed before the actual path is known has to be undo-able or without side effects somehow. This is usually done by restricting speculative execution to fetching and caching. A superscalar processor should do branch prediction because any time it is waiting for a branch to be decided, the pipeline is emptying and the functional units are not being fully utilized.

Problem 2

By tracing the execution and recording whether the branch is not taken (PC goes from 3 to 4) or is taken (PC jumps from 3 to 5), the following program flow is obtained:

```
T N T N T T N
```

Based on this, and that the branch predictor is initially set to `TT`, we get the following stream of branch predictions:

```
t t T N T N T T N
. . T T T T T T T
```

where the first line is whether the branch was actaully taken or not, and the second line is the branch predictor's output at that time. You can see that since there are never two consecutive `N`'s, the branch predictor never predicted `N` because of this particular algorithm. It makes 7 predictions, 4 of which were correct, for a total of about 57% accuracy.

Problem 3

The key to the Specter vulnerability is trying to access sensitive memory and use it as an offset to fetch some legal memory, cacheing it. Combined with exploits to detect what memory is cached, this allows for the value stored in sensitive memory to be known. This caching happens before the bounds checking is complete, which of course fails and prevents the sensitive value from being directly accessed.

It can be better understood with an example: `A[B[x]]`, where `A` and `B` are both arrays and x is our maliciosly constructed offset that will cause `B[x]` to point to potentially sensitive information. The processor has to check the bounds of the program's memory to make sure `B[x]` is in bounds, and trigger a trap if it is not (usually seen as a segfault). In the case of the exploit, `B[x]` is an illegal address, but the branch predictor assumes it is legal so that it can speculatively execute the rest of the code. This spectulative execution consists of fetching the illegal value `B[x]` and using it to fetch `A[B[x]]`, which is somewhere legal inside the programs memory. The branch predictor then discovers that `B[x]` was out of bounds, cleans up, and throws a segfault which our program catches. Then a cache detection attack (usually a timing attack or others is a high resolution timer is not accessible) is used to determine which element of `A` was accessed, and thus what the value of `B[x]` is.