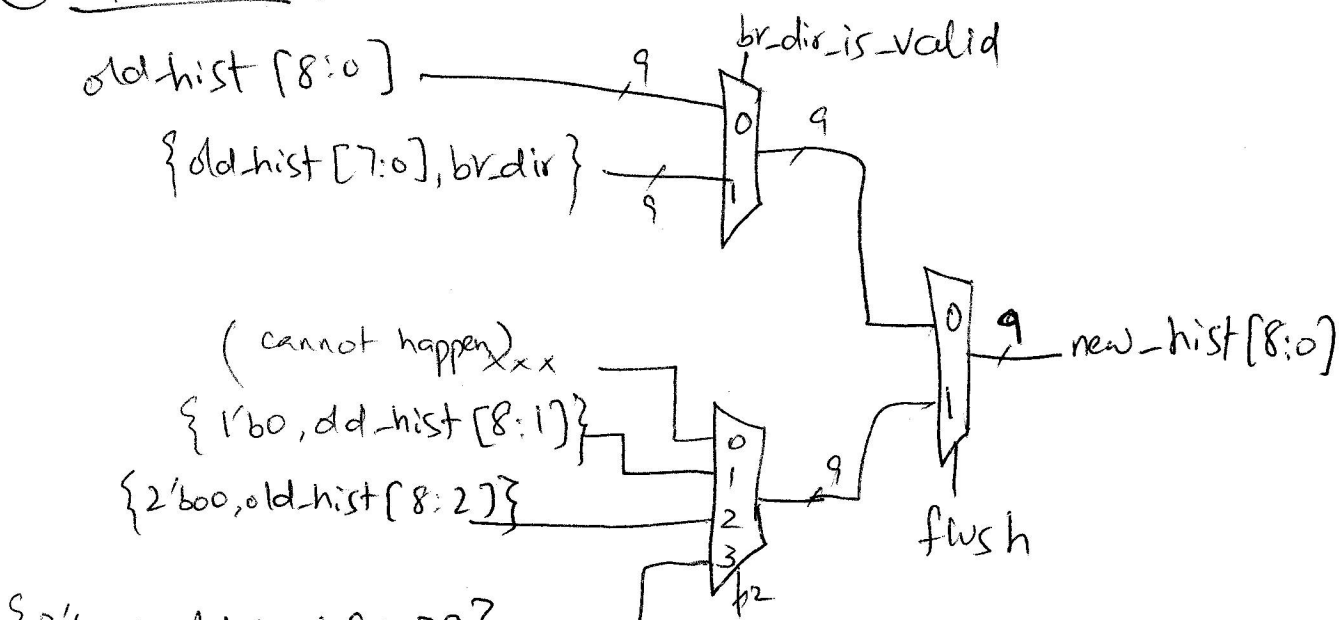
Answer 21: (a) 9-bits.

(b)



(c) Verilog code.

```
module bp_hist( ~~ports~~ ports );
input [8:0] old_hist;
input br_dir_is_valid;
input br_dir;
output [8:0] new_hist;
input flush;
input [1:0] num_br;

wire [8:0] appended, remove1, remove2, remove3,
           unflushed_val, flushed_val;

assign appended = { old_hist[7:0], br_dir };
assign remove1 = { 1'b0, old_hist[8:1] };
assign remove2 = { 2'b00, old_hist[8:2] };
assign remove3 = { 3'b000, old_hist[8:3] };

mux2$ muxA [8:0] (unflushed_val, old_hist, appended,
                  br_dir_is_valid);
mux4$ muxB[8:0] (flushed_val, old_hist, remove1,
                 remove2, remove3, num_br[0],
                 num_br[1]);
mux2$ muxC [8:0] (new_hist, unflushed_val flushed_val
```

Problem 2. The question asked which of the four is problematic. Problematic means there is not enough of it in the case of bandwidth and too long in the case of latency. Answer: on-chip bw is as large as you need it. It is unconstrained by the packaging, and near-neighbor permits almost unlimited bw. It is true that chip area is needed for non-near neighbor communication, but even that is less of a problem than the others. On the other hand, on-chip latency (getting values from one place to another at current frequencies) is a problem. Many pipelines have stages that do nothing but move information. Off-chip bw is limited by pin count. Off-chip latency, which could be handled by prefetching is only good if the off-chip bw allows it, which it usually doesn't.

Problem 3. A partial match is the situation wherein a predicted trace does not match an actual trace. With partial matching the blocks that match are issued to the execution core. With no partial matching all blocks, including those that match, are thrown away and fetching starts from the beginning at the I cache. Example: The stored trace consists of blocks A,B,C. The branches predict A,B,D. A and B match. With partial matching they are shipped to the execution core. With no partial matching, A is fetched from I cache.
Major pro: Better fetch bandwidth with partial matching.
Major con: More complex logic which could easily affect cycle time.

Problem 4. Load/Store means separate instructions are required to load from memory and to store from memory, with the result that operate instructions can only operate on data that has already been obtained from memory, and results can not be sent to memory as a result of execution of the operate instruction. The temporary benefit was that compilers could schedule loads early, hiding cache misses to source data, thereby not stalling an in-order pipeline due to a cache miss. Out-of-order execution removed that benefit by preventing cache misses from stalling the pipeline unnecessarily. Incidentally, we can have complex addressing modes in LD/ST ISAs, and short cycle times in non-LD/ST ISAs. We can have fixed or variable length in both. We can have simple or complex decode in both.

Problem 5. The structure is the BTB which is accessed at the same time as the I cache. This means a previously accessed branch instruction can provide the target address and the branch prediction in the same cycle that the branch instruction is fetched, that is, BEFORE the branch instruction is decoded. This means the PC can be loaded with the target address of the branch at the end of the cycle in which the branch instruction is fetched. ...which means no pipeline bubble waiting for decode and PC+offset addition to take place.

Problem 6. Classical VLIW requires all instructions in the long instruction word to operate in lockstep. This constraint was removed by the Decoupled Access/Execute paradigm, which fetched the two-tuple, but then split it into an access part (LD or ST) and execute part (ADD, MUL, etc) that then went to the respective in-order queues associated with access unit and the execute unit.

Problem 7. The 2-bit counter performs worst, but warms up almost immediately so it can be used (better than nothing) until the other predictors have warmed up. GAg has better accuracy, but takes longer to warm up and suffers from a lot of interference. PAp is the most accurate, suffers from essentially no interference, but takes a very long time to warm up and also requires an enormous amount of storage.

Problem 8. The compiler can "predict" the future by profiling the code on a sample data set. Two caveats are: (1) that the data set used for profiling is representative of the data set used for running for real, and (2) that the program execution does not suffer from phase changes since the result of profiling is a single value for each item examined. If that value changes as a result of phase changes, the compiler is not able to capture that to benefit at run-time.

Problem 9. Not reasonable to include an answer to this on the solution sheet since each student who did choose to answer selected a different point.

Problem 10. Actually, there are pros and cons about each. As you know, since we belabored it in class, x86 instructions have prefixes so you do not even get to the opcode until you have passed the prefixes. Also, the opcode can be one byte or two bytes. Once you have the opcode and modR/M byte, however, you are pretty much home free, since that specifies the length of the instruction. The VAX was nice in that there were no prefixes and, initially at least, all opcodes were a single byte. However, one very big negative (which to me kills all other advantages) is that each operand takes as many bytes as it needs, so the 2nd operand specifier can't even be identified until the 1st operand specifier has been decoded. The 3rd operand specifier can't be found until the 2nd operand specifier has been decoded. This has been referred to as the sequential decode problem which makes the VAX tougher to decode quickly than the x86.