Problem 1:

IF/ID:
- 32 bits for instruction
- 32 bits for pc+4

Total: 64 bits

ID/EX:
- 5 bits for WB rd
- 5 bits for WB rt
- 32 bits for sign-extended immediate
- 32 bits for read data 2
- 32 bits for read data 1
- 32 bits for pc + 4
- 4 bits for EX control (RegDst, ALUSrc, ALUOp0, ALUOp1)
- 3 bits for MEM control (MemRead, MemWrite, Branch)
- 2 bits for WB control (MemtoReg, RegWrite)

Total: 147 bits

EX/MEM:
- 5 bits for WB register
- 32 bits for memory write data
- 32 bits for ALU result
- 1 bit for Zero signal
- 32 bits for new pc result
- 3 bits for MEM control
- 2 bits for WB control

Total: 107 bits

MEM/WB:
- 5 bits for WB register
- 32 bits for ALU result
- 32 bits for memory read data
- 2 bits for WB control

Total: 71 bits

NOTE: The textbook gives 2 bits for the ALUOp in its examples, but that number is only valid for the simplified designs they give. A full implementation would require more than 2 bits. Also a full implementation would require an additional control bit for jumps in ID/EX and EX/MEM.
Problem 2:

(a) Control hazards – if predicted correctly, stalling is not needed to wait for the correct path to be determined allowing for many control hazard stalls to be removed, if predicted incorrectly, data is flushed but the time lost since it is the same as if no prediction was made – even fairly simple dynamic branch predictors can get >90% accuracy

(b) Data hazards – since data can be forwarded from the write-back and memory stages to the execute stage, instructions that would normally wait at the decode stage for the data can move forward and pick up the data they need in the execution stage, just before they actually use it – in a standard MIPS pipeline this removes nearly all data hazards with loads followed by ALU instructions being the major exception (they still require a single cycle stall)

(c) Data hazards/structural hazards – by reordering the code, data dependencies can be avoided by placing meaningful instructions that don't effect the dependency in between the dependent instructions, this optimization can remove structural hazards if there are multiple functional units – even though out of order processing can remove many hazards, it introduces its own breed of hazards that don't occur in in-order processors (WAW - write after write and WAR - write after read)

(d) Structural hazards – multiple functional units can allow multiple arithmetic operations to happen in parallel, this is especially important for floating point functional units which are often take multiple cycles to complete – additional functional units plus instruction scheduling can be a very effective combo

(e) Structural hazards – caches keep the data closer to the processor making it faster to perform memory operations instead of having to wait for the request to go all the way to main memory; because memory stalls hold up the rest of the pipeline, having caches can reduce the time the pipeline is held up for – in modern designs, requests to main memory can take 100s of cycles versus a couple for the L1 cache

Problem 3:

(a)

<table>
<thead>
<tr>
<th></th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
<th>CC9</th>
<th>CC10</th>
<th>CC11</th>
<th>CC12</th>
<th>CC13</th>
<th>CC14</th>
<th>CC15</th>
<th>CC16</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>IF</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th></th>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
<th>CC9</th>
<th>CC10</th>
<th>CC11</th>
<th>CC12</th>
<th>CC13</th>
<th>CC14</th>
<th>CC15</th>
<th>CC16</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>IF</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>IF</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) No, since the shift instruction still needs the $t1 value in the EX stage and still writes back to the $t1 register, it would provide no benefit and is effectively the same as the add in terms of performance.
Problem 4:

(a) 
0x5FFC
0x5FF8
0x5FF4
0x5FF4
0x5FF8
0x5FFC

(b) 

<table>
<thead>
<tr>
<th>Index</th>
<th>Valid</th>
<th>Tag</th>
<th>Lower Word Value</th>
<th>Upper Word Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0x3</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0x4</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0x6</td>
<td>1</td>
<td>0x17F</td>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td>0x7</td>
<td>1</td>
<td>0x17F</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) 

We miss on the first and third instruction giving $2/6 = 33\%$ miss rate

We hit on the other four instructions giving $4/6 = 67\%$ hit rate