Problem 1 (4.2):

a) i) RAW dependence on R1. Data flow.
    ii) No dependences.
    iii) WAW “maybe” dependence if 7(R1) == 200(R7). Name reuse.
    iv) Control dependence on R1.

b) i) Cannot be reordered because of the true dependence.
    ii) Compiler can schedule the two instructions in any order.
    iii) Cannot be reordered because of maybe dependence.
    iv) Cannot be reordered without control speculation or predication.
Problem 2 (4.18):

a) DSUB R1, R13, R14
   CMP.NE pT, pF=R1, R0
   (pF) DADDI, R2, R2, #1
   (pF) SD 0(R7), R2
   (pT) MUL.D F0, F0, F2
   (pT) ADD.D F0, F4
   (pT) S.D 0(R8), F0

b) Dependences Circled.

Original Code:
   DSUB R1, R13, R14
   BNEZ R1, L1
   DADDI R2, R2, #1
   SD 0(R7), R2
   J L2
L1:  MUL.D F0, F0, F2
    ADD.D F0, F4
    S.D 0(R8), F0
L2:

* There is a RAW dependence on R1 between the first two instructions.
There is a RAW dependence on R2 between the third and fourth instruction. There is a WAW and WAR dependence on F0 between the MUL.D and ADD.D instructions. There is a WAR dependence on F0 between the ADD.D and S.D instructions. In addition, there is the control dependence caused by the branch.

New Code:
   DSUB R1, R13, R14
   CMP.NE pT, pF=R1, R0
   (pF) DADDI, R2, R2, #1
   (pF) SD 0(R7), R2
   (pT) MUL.D F0, F0, F2
   (pT) ADD.D F0, F4
   (pT) S.D 0(R8), F0

* There is a RAW dependence on R1 between the DSUB and CMP.NE instructions. There is a RAW dependence on R2 between the DADDI and SD instructions. There is a WAW and WAR dependence on F0 between the MUL.D and ADD.D instructions. There is a WAR dependence on F0 between the ADD.D and S.D instructions. There are also RAW dependences on every use of the pF and pT registers.
Problem 3 (5.4):

a)  
L1 hit = 1 processor cycle = .91 ns  
L2 hit = 15 ns + 2 bus cycles  
= 15 ns + 2 * 3.76 = 22.52 ns  
Mem (Clean) = 60 ns + 4 bus cycles  
= 60 ns + 4 * 7.52 = 90.08 ns  
Mem (Dirty) = Writeback + Line Allocation  
= 60 ns + 4 bus cycles + 60 ns + 4 bus cycles  
= 120 ns + 8 * 7.52 = 180.16 ns

Average Mem Access Time For Instructions =  
= .98 * L1 hit + .016 * L2 hit + .002 * Mem(Clean) + .002 * Mem(Dirty)  
= .98 * .91 ns + .016 * 22.52 ns + .002 * 90.08 ns + .002 * 180.16 ns  
= .8918 ns + .36032 ns + .18016 ns + .36032 ns  
= 1.7926 ns

b)  
L1 hit = 1 processor cycle = .91 ns  
L2 hit = 15 ns + 1 bus cycles  
= 15 ns + 1 * 3.76 = 18.76 ns  
Mem (Clean) = 60 ns + 4 bus cycles  
= 60 ns + 4 * 7.52 = 90.08 ns  
Mem (Dirty) = Writeback + Line Allocation  
= 60 ns + 4 bus cycles + 60 ns + 4 bus cycles  
= 120 ns + 8 * 7.52 = 180.16 ns

Average Mem Access Time For Data Reads =  
= .95 * L1 hit + .04 * L2 hit + .005 * Mem(Clean) + .005 * Mem(Dirty)  
= .95 * .91 ns + .04 * 18.76 ns + .005 * 90.08 ns + .005 * 180.16 ns  
= .8645 ns + .7504 ns + .4504 ns + .9008 ns  
= 2.9661 ns

c)  
L1 hit = 1 processor cycle = .91 ns  
L2 hit = 15 ns + 1 bus cycles  
= 15 ns + 1 * 3.76 = 18.76 ns  
Mem (Clean) = 60 ns + 4 bus cycles  
= 60 ns + 4 * 7.52 = 90.08 ns  
Mem (Dirty) = Writeback + Line Allocation  
= 60 ns + 4 bus cycles + 60 ns + 4 bus cycles  
= 120 ns + 8 * 7.52 = 180.16 ns

Average Mem Access Time For Data Writes =  
= .95 * L1 hit + .05 (.04 * L2 hit + .005 * Mem(Clean) + .005 * Mem(Dirty))  
= .95 * .91 ns + .05 (.04 * 18.76 ns + .005 * 90.08 ns + .005 * 180.16 ns)  
= .8645 ns + .05 (.7504 ns + .4504 ns + .9008 ns)  
= .8645 ns + .05(2.1016 ns)  
= .96958 ns
d) 1 Processor Clock Cycle = .91 ns
   Instruction Access = 1.7926 ns = 1.97 cycles
   Data Read Access = 2.9661 ns = 3.26 cycles
   Data Write Access = .96958 ns = 1.07 cycles

   CPI = .7 + 1.97 + .2(3.26) + .05(1.07) = 3.38 cycles per instruction

e) Instruction Access = .98 * .48 ns + .016 * 22.52 ns + .002 * 90.08 ns +
   .002 * 180.16 ns
   = .4704 + .36032 ns + .18016 ns + .36032 ns
   = 1.3712 ns
   Data Read = .95 * .48 ns + .04 * 18.76 ns + .005 * 90.08 ns + .005 *
   180.16 ns
   = .456 + .7504 ns + .4504 ns + .9008 ns
   = 2.5576 ns
   Data Write = .95 * .48 ns + .05 (.04 * 18.76 ns + .005 * 90.08 ns + .005 *
   180.16 ns)
   = .456 + .05 (.7504 ns + .4504 ns + .9008 ns)
   = .456 + .05(2.1016 ns)
   = .56108 ns

   1 Processor Clock Cycle = .48 ns
   Instruction Access = 1.3712 ns = 2.86 cycles
   Data Read Access = 2.5576 ns = 5.33 cycles
   Data Write Access = .56108 ns = 1.17 cycles

   CPI = .7 + 2.86 + .2(5.33) + .05(1.17) = 4.68 cycles per instruction

   Old Machine: 3.38 * .91 ns = 3.08 ns per instruction
   New Machine: 4.68 * .48 ns = 2.25 ns per instruction

   New machine is 37% faster than the old machine when the clock is sped
   up by 91%.
Problem 4 (5.18):

a) There is no need to do this in the instruction cache because there are no writes and therefore there can be no coherence problem that is caused by virtual address aliasing. This problem is important in data caches because one of the copies will get updated on a write and a following load must return the newest copy. The only problem presented by virtual address aliasing in the instruction cache is cache pollution, but this will only affect performance and is not a correctness issue.

b) In this situation you would need 28 additional tag bits for the virtual address cache plus 8 bits for the process ID. So the overhead is 36 bits * number of blocks in the cache. If the block size was doubled and the cache size remained the same, the overhead would be cut in half because there would be half as many blocks. If you doubled the block size and left the number of blocks the same (doubled the size of the cache), the overhead per block would remain the same, but the percentage of overhead bits compared to the size of the cache would be cut in half. So, increasing the block size is a win either way.

Problem 5:

Write-Through: The advantages of a write-through cache are that they are easier to implement than write-back, clean and dirty state bits are not needed, and coherency is not an issue for I/O (DMA).

Write-Back: The major advantage of a write-back cache is that it saves write bandwidth on write hits and that write hits are done at the speed of the L1.

Two options on a write miss:
Write-Allocate: block brought into cache on write miss and then do write hit action
Write-No-Allocate: data just written lower level

Write-Allocate is usually used with write-back caches with the hope that subsequent writes to the block will be captured by the L1. Write-No-Allocate is usually used with write-through caches because nothing is gained by bringing the block into the L1 because all subsequent writes have to go to the lower level anyway.