ative energy consumption is (5+10bf)IC/(5IC) = 1+2bf.
• Always predicted taken
When a branch is taken five units of energy are wasted. When a branch is untaken, ten units of energy are wasted. So the total energy consumed is (5+5bf+10b(1-f))IC= (5+10b-5bf)IC and the relative energy consumption is (5+10b-5bf)/5 = 1+2b-bf
• With compiler hints and X=0.95
The total energy is: (5+0.95*5bf+0.05*10bf+0.05*10b(1-f))IC = (5+4.75bf+0.5b)IC and the relative energy consumption is (5+4.75bf+0.5b)/5 = 1+0.95bf+0.1b.

Problem 3.8

a. RAW hazards.
Because we have full forwarding, an integer instruction in ID can get an operand from EX/ME or ME/WB(int). However, if a load is in EX, a dependent integer instruction in ID must stall. Integer instructions get their operands from integer registers, and so they do not check FP pipeline registers. If an FP arithmetic instruction is in ID, we need to compare source registers to the destination of each pipeline registers in FP.
-- If an integer arithmetic/logic/store instruction or load is in ID, check ID/EX for a load
-- If an FP arithmetic instruction is in ID, check ID/EX (for FP loads), ID/FP, FP1/FP2, FP2/FP3, and FP3/FP4.
-- If an FP store is in ID, check ID/EX (for loads returning address register or FP value), FP1/FP2, FP2/FP3, and FP3/FP4.

b. For WAW hazards,
-- If an integer arithmetic/logic/store or integer load is in ID: No check is necessary, since these instructions update integer registers only and follow the integer pipeline path in process order.
-- If an FP load is in ID, check ID/FP, and FP1/FP2 and FP2/FP3 (no need to check FP3/FP4 because an FP load and an FP arithmetic instruction cannot reach the FP register file in the same cycle. This is done in ID to prevent structural hazards on the write port of the FP register file.)
-- If an FP arithmetic instruction is in ID, there is no need to check any pipeline stage because FP arithmetic instructions cannot bypass a previous instruction.
-- If an FP store is in ID, there is no need to check any pipeline stage because stores do not write value in registers.

Problem 3.9

a. For RAW hazards,
-- If an integer arithmetic/logic/store instruction or a load is in ID, check ID/EX1 (for integer instructions and loads), EX1/EX2 (for loads) and EX2/ME1 (for loads).
-- If an FP arithmetic instruction is in ID, check ID/EX1 (for FP loads), EX1/EX2 (for FP loads), EX2/ME1 (for FP loads), ID/FP, FP1/FP2, FP2/FP3 and FP3/FP4.
-- If an FP store instruction is in ID, check ID/EX1 (for loads returning address or FP value), EX/EX2 (for loads returning address or FP value), EX2/ME1(for loads returning address or FP value), ID/FP, FP1/FP2, FP2/FP3, and FP3/FP4.
b. for WAW hazards,
-- If an integer arithmetic/logic/store instruction or integer load is in ID, no check is necessary (as all integer instructions follow the same path)
-- If an FP load is in ID, check ID/FP1 and FP1/FP2. There is no need to check FP2/FP3 because logic in ID prevents two write to the same register file in the same clock.
-- If an FP arithmetic instruction is in ID, no check is necessary because FP arithmetic instructions take the longest path and all other preceding instructions are always completed before an FP arith-

metic instruction in ID.
-- If an FP store is in ID, no check is necessary because stores do not write to registers.

Problem 3.10

a. First, note that only one value (integer or FP) may be propagated in any one stage of the pipeline. For example we cannot have one instruction in EX and one instruction in FP1 in the same clock. This is because instructions are issued one at a time.

We consider values forwarded to EX first. Because integer instructions also traverse FP pipeline stages FP3, FP4 and FP5, values may be forwarded from their pipeline register to the EX stage.
EX/ME --> EX
ME/FP3 --> EX (this also include FP values forwarded by FP loads to FP stores)
FP3/FP4 --> EX
FP4/FP5 --> EX
FP5/WB - ->EX

Next we consider values forwarded to FP1.
ME/FP3 --> FP1
FP5/WB-->FP1
In this solution, we simplify by not providing a forwarding path from FP loads in FP4 and in FP5 to FP1. Rather we stall. One could also forward FP load values (ONLY) from FP3/FP4 and FP4/FP5 to avoid stalling on them.

b. Remaining RAW hazards
-- If an integer arithmetic/logic/store/load instruction is in ID, check ID/EX (for preceding loads)
-- If an FP arithmetic instruction is in ID, check ID/EX (for preceding loads), ID/FP1, FP1/FP2, FP2/FP3, FP3/FP4.
-- If an FP store instruction is in ID, check ID/EX (for loads returning address or FP value), ID/FP1, FP1/FP2, FP2/FP3, FP3/FP4 (for FP values).

c. Stores update machine state early (this is particularly bad because stores update memory). One simple solution is to stall stores in ID until it is determined that no previous instruction currently in the pipeline can trigger an exception.

Problem 3.11

a. In ID2 the two instructions have no dependency and are a correct pair (with possibly one of them a NOOP, which is fine). Each can execute as soon as they don't have any data hazard (RAW or WAW) with previous instructions currently in the pipeline. To respect process order, the lower instruction cannot start execution before the upper instruction. So the HDU associated with ID2 simply checks for hazards with instructions currently in the pipeline for both instructions in ID2 (in the scalar processor checks were made for a single instruction, now it is two, that's the only difference).

b. A branch is resolved in EX. Because branches are always predicted untaken, all the following instructions are in IF, ID1, ID2 and FP1. Thus, when a branch is taken, we need to flush the instructions in all these stages (and no more), a total of 7 instructions.

c. We show the code as it will be presented to ID2 by ID1. Instructions between parenthesis are not instructions in the code, rather they have been "inserted" by ID1. In the right most column, the

number of cycles associated with each instruction pair in ID2 is shown (1 cycle in ID2 plus the stalls due to data dependencies based on the operation latencies in Figure 2). In the case of the taken branch we add 3 cycles because IF, ID1 and ID2 are flushed.

	Upper ID2	Lower ID2	Cycles
LOOP	L.D F2,0(R1)	(NOOP)	(1)
	(NOOP)	ADD.D F4,F2,F4	(2)
	L.D F6, -8(R1)	(NOOP)	(1)
	(NOOP)	ADD.D F8,F6,F4	(4)
	S.D F8, 0(R1) (4)	(NOOP)	(5)
	SUBI R1,R1,16(1)	(NOOP)	(1)
	BNEZ R1, LOOP(3)	(NOOP)	(1+3)

The execution time of one iteration of the loop is 18 cycles

For the machine in Exercise 3.8, the numbers between parenthesis are the number of cycles in ID and branch penalties due to flushing.

```
LOOP    L.D F2,0(R1)    (1)
        ADD.D F4,F2,F4  (2)
        L.D F6, -8(R1)  (1)
        ADD.D F8,F6,F4  (4)
        S.D F8, 0(R1)  (5)
        SUBI R1,R1,16  (1)
        BNEZ R1, LOOP  (1+2)
```

Therefore, it takes 17 cycles for one iteration of the loop.

For the machine in Exercise 3.9, the latency of operation of a Load is now 3. The operation latency of a reg-to-reg instruction is now 1. The operation latency of FP arithmetic instruction is unchanged (4). A branch is resolved in EX1 and 3 stages must be flushed when it is taken.

```
LOOP    L.D F2,0(R1)    (1)
        ADD.D F4,F2,F4  (4)
        L.D F6, -8(R1)  (1)
        ADD.D F8,F6,F4  (4)
        S.D F8, 0(R1)   (5)
        SUBI R1,R1,16   (1)
        BNEZ R1, LOOP   (2+3)
```

Therefore each iteration takes 21 cycles. However the clock rate can be raised by pipelining bottleneck stages (such as IF, EX and ME).

Conclusion: In terms of cycles, no machine has a clear advantage. The superpipeline has an advantage if it can be clocked faster. The superscalar pipeline is penalized because the code does not use the two pipelines effectively.

Problem 3.12

a. The three penalties (data, branch, and memory) measured in cycles increase proportionally to the number of stages, assuming that the increase in stages is spread evenly among the 5 fundamental functions of the 5 stage pipeline --IF,ID,EX,ME, and WB. This may be seen as a gross approximation, but we are not looking for an exact value, just a trend.

b. The CPI of the pipeline with K stages is given by:

$$CPI(K) = 1 + (\alpha_d + 2\alpha_b + \alpha_m)\frac{K}{5} = 1 + A \times K$$

The average execution time of one instruction is equal to

$$CPI(K) \times T_K = (1 + A \times K)\left(\frac{T}{K} + t_l\right)$$