Problem 3.7

In the classic 5-stage pipeline, it is proposed to predict branches as always taken instead of always untaken. The branch instruction is decoded in ID and its target address is computed in ID. At the end of ID, a conditional branch is always taken and IF is systematically flushed. Then, in the EX stage, the branch condition is evaluated. If the branch is verified taken, then execution continues. However, if the branch is verified untaken, the IF and ID stages are flushed and the instruction at Branch_PC+4 is fetched.

a. What is the fraction f of branches that should be taken so that the design with branch predicted always taken is a good choice over branch predicted always untaken?

b. A hint bit is associated with each conditional branch instruction. The compiler sets the hint bit to steer the hardware prediction to "taken" and it resets the hint bit to steer the hardware prediction to "untaken". The hint bit is known in the decode stage so that the two hardwired schemes (always taken and always untaken) can be applied with no additional loss of cycle to each branch instruction. What should be the success rate of the compiler's prediction so that the performance of this approach is always better than the hardware scheme where a branch is predicted always taken? What should be the success rate of the compiler's prediction so that the performance of this approach is always better than the hardware scheme where a branch is predicted always untaken? The compiler prediction success rates for taken and untaken branches are assumed equal. Please use the following variables in your solution:

- f is the fraction of taken branches
- X is the success rate of the compiler prediction algorithm (i.e., the fraction of branches that are accurately predicted by the compiler to meet the conditions). X should be a function of f in both cases.

c. Take the 5-stage pipeline with perfect branch handling (optimum, no cycle ever wasted on branches) as the baseline. Compare the energy per instruction (EPI) of this baseline with the following cases:

- Always predicted untaken
- Always predicted taken
- Compiler-based prediction with hint bit and 5% misprediction rate uniform over all predictions.
-

To make this problem possible we assume that each stage of the pipeline consumes the same energy per clock (whatever the instruction) and that the energy needed to flush a stage is negligible. Also assume that the fraction of instructions that are branches is b.

Problem 3.8

In this problem and the next two problems we evaluate the hardware needed to detect hazards in various static pipelines with out of order instruction execution completion. We consider the floating-point extension to the 5-stage pipeline, displayed in Figure 3.8.

Each pipeline register carries its destination register number, either floating-point or integer. ME/WB carries two instructions, one from the integer pipeline and one from the floating-point arithmetic pipeline.

Consider the following types of instructions consecutively:

• Integer arithmetic/logic/Store instructions (inputs: two integer registers) and all Load instructions (input: one integer register)
• Floating-point arithmetic instructions (inputs: two floating-point registers)
• Floating-point Stores (inputs: one integer and one floating-point registers).

All values are forwarded as early as possible. Both register files are internally forwarded. All data hazards are resolved in the ID stage with a hazard detection unit (HDU). ID fetches registers from the integer and/or from the floating-point register file, as needed. The opcode selects the register file from which operands are fetched (S.D fetches from both)

a. To solve RAW data hazards on registers (integer and/or floating-point), hardware checks (interlocks) between the current instruction in ID and instructions in the pipeline may stall the instruction in ID. List first all **pipeline registers** that **must** be checked in ID. Since ME/WB may have two destination registers list them as ME/WB(int) or ME/WB(fp). Please don't list pipeline stages, list pipeline registers. Please make sure that the set of checks is minimum.

b. To solve WAW hazards on registers, we check the destination register in ID with the destination register of instructions in various pipeline stages. Please list the pipeline registers that must be checked. Make sure that the set of checks is minimum. IMPORTANT: remember that there is a mechanism in ID to avoid structural hazards on the write register ports of both register files.

Your solutions specifying the hazard detection logic should be written as follows for both RAW and WAW hazards:

--If Integer arithmetic/logic/Store/Load instruction in ID check <pipeline registers>
--If FP Load instruction in ID check <pipeline registers>
--If FP arithmetic instruction in ID check <pipeline registers>
--If FP Store instruction in ID check <pipeline registers>

Problem 3.9

Repeat Problem 3.8 for the superpipelined architecture in Figure 3.10. Assume forwarding for all instructions including FP Stores. Note that both floating-point and integer values can now be forwarded from both ME1/ME2 and ME2/WB.

Your solutions specifying the hazard detection logic should be written as follows for both RAW and WAW hazards:

--If Integer arithmetic/logic/Store instruction or Load instruction in ID check <pipeline registers>
--If FP Load instruction in ID check <pipeline registers>
--If FP arithmetic instruction in ID check <pipeline registers>
--If FP Store instruction in ID check <pipeline registers>

Problem 3.10

In the pipeline of Figure 3.9 WAW data hazards on registers are eliminated and exceptions can be handled in the WB stage where instructions complete in process order as in the classic 5-stage pipeline. As always values are forwarded to the input of the execution units.

a. List all required forwarding paths from pipeline registers to either EX or FP1 to fully forward values for all instructions. List them as source-->destination (e.g, FP2/FP1-->FP1)

b. Given those forwarding paths, indicate all checks that must be done in the hazard detection unit associated with ID to solve RAW hazards.
Your solutions specifying the hazard detection logic should be written as follows for RAW hazards on registers:

--If Integer arithmetic/logic/Store or Load instruction in ID check <pipeline registers>
--If FP arithmetic instruction in ID check <pipeline registers>
--If FP Store instruction in ID check <pipeline registers>

c. This architecture still has a subtle problem with respect to exception handling. Namely Stores are executed early and modify memory before they retire in the write-back stage. What is the problem. Can you propose a solution to this problem? (Please do not propose the solution of saving the memory value and then restoring it upon an exception.)

Problem 3.11

Consider the superscalar architecture of Figure 3.45. Two consecutive instructions are fetched at a time, incrementing PC by 8. To simplify pipeline interlocks, we split the decode stage into two stages ID1 and ID2. A switch with two settings (*straight* and *across*) separates ID1 and ID2. Upper ID2 must be an integer/branch instruction or an FP Load/Store. Lower ID2 must be an arithmetic FP instruction.

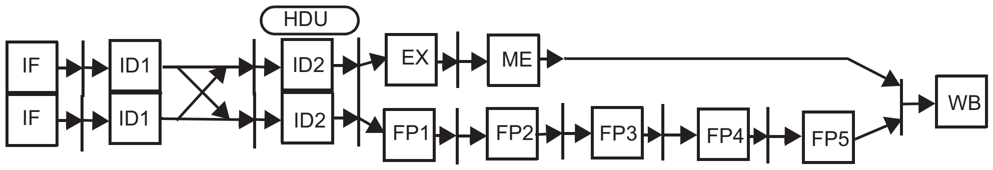

Figure 3.45. Two-way superscalar CPU

Let I1 be the upper instruction and I2 be the lower instruction in ID1. I1 must proceed to ID2 before or at the same time as I2 is allowed to proceed in order to adhere to process order. The following is done in ID1:

• If I1 is an integer/branch instruction or an FP Load/Store or a NOOP and I2 does not depend on I1 (the expected case) then set switch to straight.
• If I1 is an FP Load and I2 is an instruction using the value returned by the Load, stall I2 in ID1 and move I1 to ID2 with switch set to straight (Lower ID2 is NOOPed).
• If I1 is an FP arithmetic instruction, stall I2, and move I1 to ID2 with switch set to across (Upper ID2 is NOOPed).
• If I1 and I2 are both an integer/branch instruction or an FP Load/Store, stall I2 in ID1 and move I1 to ID2 with switch set to straight.(Lower ID2 is NOOPed)
• If I2 is an integer/branch instruction or an FP Load/Store and I1 is a NOOP, move I1, I2 to ID2 with switch set to across.

Thus if the two fetched instructions are dependent or are the wrong pair, they are serialized in ID1. Instructions in ID2 are subject to stalls due to pipeline hazard as in the single issue processor and proceed if they have no data hazard with previous instructions still in the pipeline. We deploy the same forwarding paths as in Figure 3.8. When instruction(s) are stalled in ID2, then instructions in IF and ID1 are stalled as well.

a. Describe briefly the function of the HDU associated with ID2

b. Explain how a branch is processed (consider both cases when the branch is upper or lower in ID1), assuming that branches are always predicted untaken by the hardware.

c. Consider the following code:

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

Compare the execution times of one iteration of this loop (not the last iteration) on this machine and the machines of Problem 3.8 and 3.9. For the machine of Problem 3.9 assume that branches can be resolved in EX1.

Problem 3.12

It would seem that superpipelining is a scalable solution to providing higher performance by deeply pipelining and increasing the clock rate. However there are three impediments to this:

• As the number of stages increases the functional logic delay decreases proportionally but the delay of the pipeline registers does not change
• The penalties (bubbles) in cycles caused by data dependencies increase
• The number of stages to flush on a mispredicted branch increases
• The penalty of cache misses increases as memory is not improved by deeper pipelines.

We model these effects as follows. Let T be the clock period of the single-cycle CPU. Let K be the number of stages in the pipeline. The clock cycle of the pipelined CPU with K stages is modeled as:

$$T_K = \frac{T}{K} + t_l = \frac{1}{f_K}$$

where t_l is the time needed to latch the output of each stage (setup time).

The penalty per instruction (in cycles) due to data hazards is modeled as:

$$\Delta_{data} = \alpha_d \frac{K}{5}$$

Similarly the penalty per instruction due to mispredicted branches (control hazards) is modeled as:

$$\Delta_{branches} = 2\alpha_b \frac{K}{5}$$

Finally, cache miss penalty also affects the speedup of deep pipelines because more cycles are needed to resolve cache misses. This can be modeled as:

$$\Delta_{memory} = \alpha_m \frac{K}{5}$$

α_d is approximately the average number of stalls per instruction in ID because of data hazard in the 5 stage pipeline, α_b is the fraction of instructions that are mispredicted branches (assuming a 2 clock penalty in the 5-stage pipeline), and α_m is the average number of cycles wasted by cache misses per instruction in the 5-stage pipeline.