1. A multiprocessor machine has 1024 processors. On this machine we map a computation in which N iterate values must be computed and then exchanged between the processors. Values are broadcast on a bus after each iteration. Each iteration proceeds in two phases. In the first phase each processor computes a subset of the N iterates. Each processor is assigned the computation of K = N/P iterates, where P is the number of processors involved. In the second communication phase each processor broadcasts its results to all other processors, one by one. Every processor waits for the end of the communication phase before starting a new computation phase. Let Tc be the time to compute one iterate and let Tb be the time to broadcast one value on the bus. We define the computation-to-communication ratio R as Tc/Tb. Note that, when P = 1, no communication is required.

At first, we use the premise of Amdahl's speedup (i.e., the same workload spread across an increasing number of processors). Under these conditions:
(a) Compute the speedup as a function of P and R.

(b) Compute the maximum possible speedup as a function of P and R, for K = 1, 2, . . . , 1024.

(c) Compute the minimum number of processors needed to reach a speedup greater than 1 as a function of P and R.

Second, we use the premise of Gustafson's law, namely that the uniprocessor workload grows with the number of processors so that the execution time on the multiprocessor is the same as that on the uniprocessor. Assume that the uniprocessor workload computes 1024 iterates.
(d) What should the size of the workload be (as a number of iterates) when P processors are used, as a function of P and R? Pick the next closest integer value for the number of iterates.

2. This problem is about the difficulty of reporting average speedup numbers for a set of programs or benchmarks. We consider four ways of reporting average speedups of several programs:

- take the ratio of average execution times, S1;
- take the arithmetic means of speedups, S2;
- take the harmonic means of speedups, S3;
- take the geometric means of speedups, S4.

Consider the two improvements of a base machine, one improving floating-point performance and one improving memory performance: Two improvements are considered to a base machine with a load/store ISA and in which floating-point arithmetic instructions are implemented by software handlers. The first improvement is to add hardware floating-point arithmetic units to speed up floating-point arithmetic instructions. It is estimated that the time taken by each floating-point instruction can be reduced by a factor of 10 with the new hardware. The second improvement is to add more first-level data cache to speed up the execution of loads and stores. It is estimated that, with the same amount of additional on-chip cache real-estate as for the floating-point units, loads and stores can be speeded up by a factor of 2 over the base machine.

Three programs are simulated: one with no floating-point operations (Program 1), one dominated by floating point operations (Program 2), and one with balance between memory accesses and floating point operations (Program 3). The execution time of each program on the three machines is given in Table 1:

Table 1 Execution times of three programs

Machines	Program1	Program2	Program3
Base machine	1s	10ms	10s
Base + FP units	1s	2ms	6s
Base + Cache	0.7s	9ms	8s

(a) Compute the average speedups to the base machine for each improvement. Which conclusions would you draw about the best improvement if you were to consider each average speedup individually?

(b) To remove the bias due to the difference in execution times, we first normalize the execution time of the base machine to 1, yielding the normalized execution times in Table 2. Compute the four average speedups to the base machine for both improvements. Which conclusions would you draw about the best improvement if you were to consider each average speedup individually?

Table 2 Normalized execution times of three programs

Machines	Program1	Program2	Program3
Base machine	1	1	1
Base + FP units	1	0.2	0.6
Base + Cache	0.7	0.9	0.8

3. In this problem 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 1.

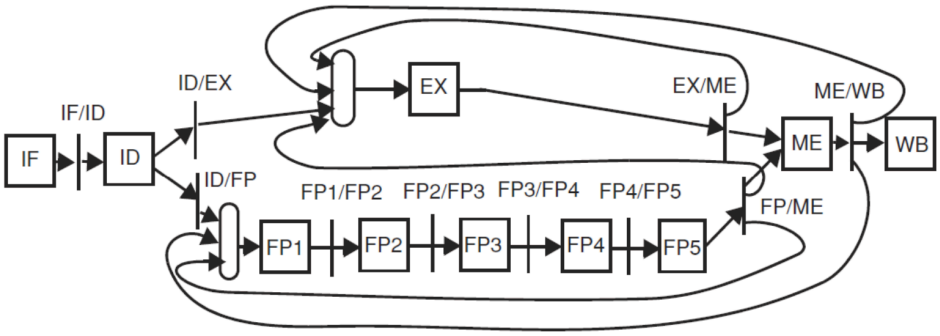

Figure 1 Pipeline with out-of-order execution completion

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 register).

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). Note that stores need both inputs in EX at the latest.

(a) To solve RAW data hazards on registers (integer and/or floating-point), hardware checks 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). Do not list pipeline stages, list pipeline registers, and 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. List the pipeline registers that must be checked. Make sure that the set of checks is minimum. Important: remember that there is already a mechanism in the pipeline design to avoid structural hazards on the write register ports of both register files, so it is not possible for two instructions that write to the same register to reach WB in the same cycle.

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 >.

4. In the pipeline of Figure 2, 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.

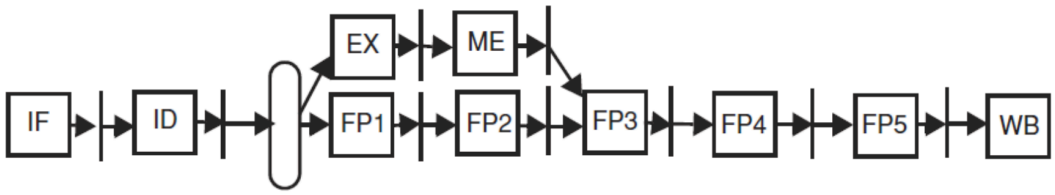

Figure 2 Forcing in-order completion in the WB stage.

(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>.

5. In this problem we compare the performance of four dynamically scheduled processor architectures on a simple piece of code computing Y = Y∗X + Z, where X, Y, and Z are (double-precision–8 bytes) floating-point vectors.
Using the core ISA of Table 0 in the Appendix, the loop body can be compiled as follows:

```
LOOP   L.D F0,0(R1)        /X[i] loaded in F0
       L.D F2,0(R2)        /Y[i] loaded in F2
       L.D F4,0(R3)        /Z[i] loaded in F4
       MUL.D F6,F2,F0      /Multiply X by Y
       ADD.D F8,F6,F4      /Add Z
       ADDI R1,R1,#8       /update address registers
       ADDI R2,R2,#8
       ADDI R3,R3,#8
       S.D F8, -8(R2)      /store in Y[i]
       BNE R4,R2,LOOP/     /(R4)-8 points to the last element of Y
```

The initial values in R1, R2, and R3 are such that the values are never equal during the entire execution. (This is important for memory disambiguation.) The architectures are given in Figures 3, 4, 5 and 6, and the same parameters apply. Branch BNE is always predicted **taken** (except in the basic Tomasulo, where branches are not predicted at all and stall in the dispatch stage until their outcome is known).

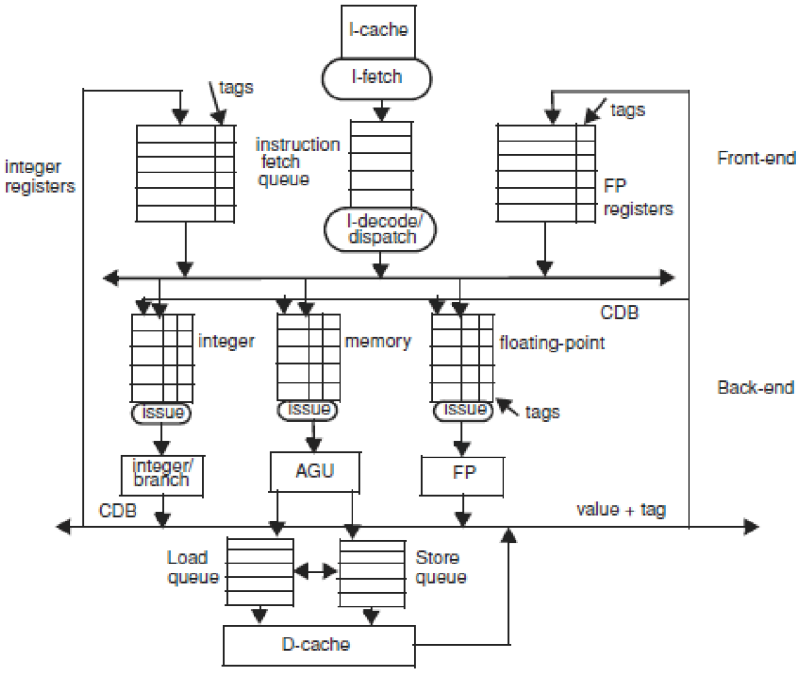

Figure 3 Hardware for basic Tomasulo algorithm.

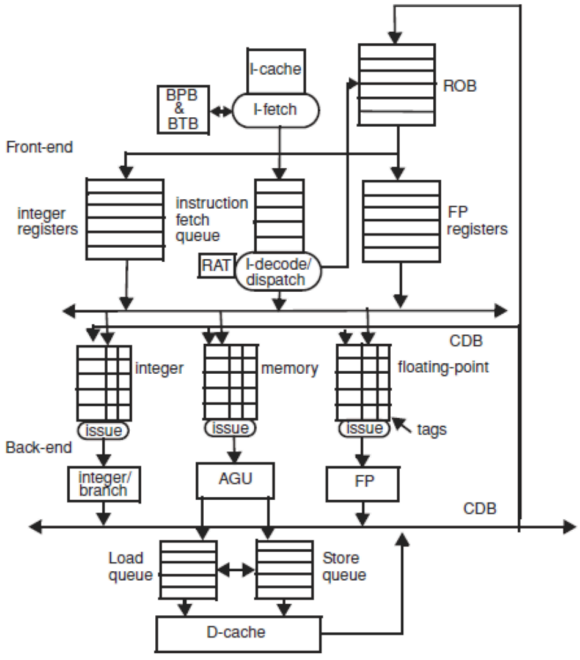

Figure 4 Tomasulo algorithm with support for speculative execution.

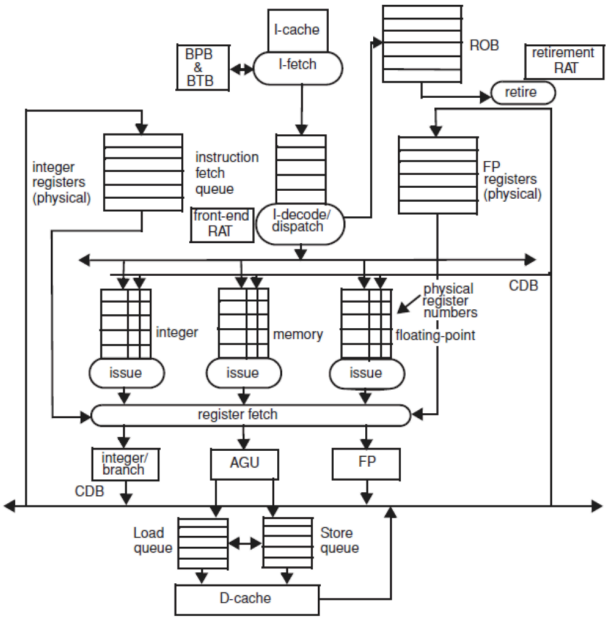

Figure 5 Speculative microarchitecture with explicit register renaming and register fetch after issue.

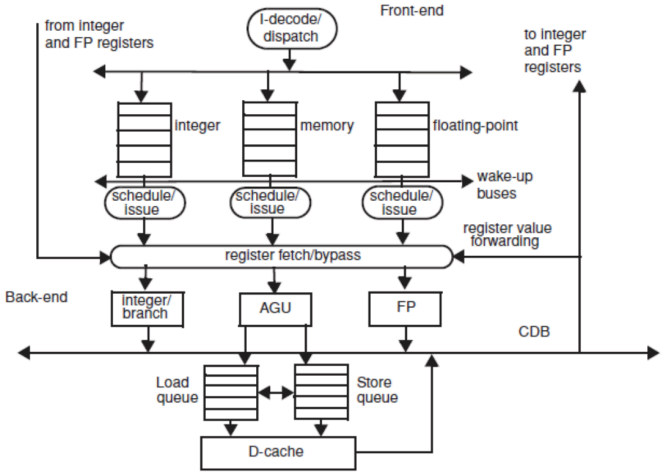

Figure 5 Back-end for a speculative microarchitecture with speculative scheduling.

Keep in mind the following important rules (whenever they apply):

- instructions are always fetched, decoded, and dispatched in process order;
- in speculative architectures, instructions always retire in process order;
- in speculative architectures, stores must wait until they reach the top of the ROB before they can issue to cache (i.e. a store issues to cache in the cycle after the previous instruction retires).
- The solution should include **two** loads from the next iteration of the loop

(a) Tomasulo algorithm – no speculation. Please fill a table like Table 3 clock-by-clock for the **first iteration of the loop**. Each entry should be the clock number when the event occurs, starting with clock cycle 1. Add comments as you see fit.

Table 3 Tomasulo algorithm – no speculation

		Dispatch	Issue	Exec Start	Exec complete	Cache	CDB	Comments
I1	L.D F0,0(R1)							
I2	L.D F2,0(R2)							

(b) Tomasulo algorithm with speculation. Please fill a table like Table 4 clock-by-clock for the first iteration of the loop. Each entry should be the clock number when the event occurs, starting with clock cycle 1. Please be attentive to the fact that (contrary to Tomasulo with no speculation) stores cannot execute in cache until they reach the top of the ROB. Also branches are now predicted taken.

Table 4 Tomasulo algorithm – with speculation

		Dispatch	Issue	Exec Start	Exec complete	Cache	CDB	Retire	Comments
I1	L.D F0,0(R1)								
I2	L.D F2,0(R2)								

(c)) Tomasulo algorithm with speculation and register fetch after issue. Please fill a table like Table 5 clock-by-clock for the first iteration of the loop. Each entry should be the clock number when the event occurs, starting with clock 1.

Table 5 Speculation and Register Fetch after Issue

		Dispatch	Issue	Register Fetch	Exec Start	Exec complete	Cache	CDB	Retire	Comments
I1	L.D F0,0(R1)									
I2	L.D F2,0(R2)									

(d) Speculative scheduling. Please fill a table like Table 6 clock-by-clock for the first iteration of the loop. Each entry should be the clock number when the event occurs, starting with clock 1.

Table 6 Speculative Scheduling

		Dispatch	Issue	Register Fetch	Exec Start	Exec complete	Cache	CDB	Retire	Comments
I1	L.D F0,0(R1)									
I2	L.D F2,0(R2)									

Appendix

Table 0: Core ISA

Types	Opcode	Assembly code	Meaning	Comments
Data transfers	LB, LH, LW, LD	LW R1,#20(R2)	R1<=MEM [(R2)+20]	for bytes, half-words, words, and double words
	SB, SH, SW, SD	SW R1,#20(R2)	MEM[(R2)+20] <= (R1)	
	L.S, L.D	L.S F0,#20(R2)	F0<=MEM [(R2)+20]	single/double float load
	S.S, S.D	S.S F0,#20(R2)	MEM[(R2)+20] <= (F0)	single/double float store
ALU operations	ADD, SUB, ADDU, SUBU	ADD R1,R2,R3	R1<=(R2)+(R3)	addition/subtraction signed or unsigned
	ADDI, SUBI, ADDIU, SUBIU	ADDI R1,R2,#3	R1<=(R2)+3	addition/subtraction immediate signed or unsigned
	AND, OR, XOR	AND R1,R2,R3	R1<=(R2).AND. (R3)	bit-wise logical AND, OR, XOR
	ANDI, ORI, XORI	ANDI R1,R2,#4	R1<=(R2).ANDI.4	bit-wise AND, OR, XOR immediate
	SLT, SLTU	SLT R1,R2,R3	R1<=1 if R2<R3 else R1<=0	test R2,R3, outcome in R1, signed or unsigned comparison
	SLTI, SLTUI	SLTI R1,R2,#4	R1<=1 if R2<4 else R1<=0	test R2, outcome in R1, signed or unsigned comparison
Branches/jumps	BEQZ, BNEZ	BEQZ R1,label	PC<=label if (R1)=0	conditional branch-equal 0/not equal 0
	BEQ, BNE	BNE R1,R2, label	PC<=label if (R1)=(R2)	conditional branch-equal/not equal
	J	J target	PC<=target	target is an immediate field
	JR	JR R1	PC<=(R1)	target is in register
	JAL	JAL target	R1<=(PC)+4; PC<=target	jump to target after saving the return address in R31
Floating-point	ADD.S,SUB.S, MUL.S,DIV.S	ADD.S F1,F2,F3	F1<=(F2)+(F3)	float arithmetic single precision
	ADD.D,SUB.D, MUL.D,DIV.D	ADD.D F0,F2,F4	F0<=(F2)+(F4)	float arithmetic double precision